A Modular Mixed Signal VLSI Design Approach for Digital Radar Applications

THESIS

Brian M. Brakus, Captain, USAF

AFIT/GCE/ENG/07-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/07-02

A Modular Mixed Signal VLSI Design Approach for Digital Radar Applications

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Brian M. Brakus, B.S.Cp.E.

Captain, USAF

March 2007

AFIT/GCE/ENG/07-02

A Modular Mixed Signal VLSI Design Approach for Digital Radar Applications

Brian M. Brakus, B.S.Cp.E.

Captain, USAF

Approved:

AFIT/GCE/ENG/07-02

Abstract

This study explores the idea of building a library of VHDL configurable components for use in digital radar applications. Configurable components allow a designer to choose which components he or she needs and to configure those components for a specific application. By doing this, design time for ASICs and FPGAs is shortened because the components are already designed and tested. This idea is demonstrated with a configurable dynamic pipelinable fast fourier transform. Many FFT implementations exist, but this implementation is both configurable and dynamic. Pre-synthesis customization allows the FFT to be tailored to almost any DSP application, and the dynamic property allows the FFT to calculate different length FFTs real-time. Three objectives will be accomplished: design and characterization of the aforementioned FFT; analysis of the error involved in the FFT calculation using different twiddle factor bit widths; and finally an analysis of all the configurations for the synthesized design using a 90 nm technology library. Speeds of up to 225 MHz have been simulated for a length-1024 FFT using the 90 nm technology.

Acknowledgements

First and foremost, I owe a large debt of gratitude to my wife for her understanding and patience during this research. Additionally, I would like to thank Dr. Yong Kim for his wisdom and helpfulness with the research and writing of this thesis. I would also like to thank Dr. Marty Emmert and Dr. Greg Creech for allowing me to work with AFRL/SND and I hope the knowledge and data within this paper is very useful. Lastly, I'd like to thank my fellow students for their knowledge and research efforts: Jason Paul, Joe Pomager, and Jason Shirley.

Brian M. Brakus

Table of Contents

Page

List of Figures

Figure Page

List of Tables

List of Abbreviations

Abbreviation		Page
EW	Electronic Warfare	1
DSP	Digital Signal Processing	1
ADC	Analog to Digital Converter	1
ASIC	Application Specific Integrated Circuit	1
RTL	Register Transfer Language	1
FPGA	Field Programmable Gate Array	2
DoD	Department of Defense	2
FFT	Fast Fourier Transform	2
TSMC	Taiwan Semiconductor Manufacturing Company	3
VHSIC	Very-High-Speed Integrated Circuit	3
VHDL	VHSIC Hardware Descriptor Language	3
AFRL	Air Force Research Lab	4
DFT	Discrete Fourier Transform	6
DIT	Decimation In Time	7
DIF	Decimation In Frequency	7
CMOS	Complementary Metal-Oxide Semiconductor	16
DITF	Decimation-In-Time-Frequency	55

A Modular Mixed Signal VLSI Design Approach for Digital Radar Applications

I. Introduction

Since at least the year 2000 the Pentagon has seen a need for highly advanced Electronic warfare (EW) aircraft. The Pentagon published a Kosovo after-action report to Congress discussing how NATO forces had difficulty in targeting missile sites [8]. Also, a separate report said the problems included interference from other aircrafts' jammers with friendly targeting devices. These reports preempted Congress to begin a study in ways to improve EW. Billions of dollars are spent researching and developing newer and more advanced radar systems. In addition to the high costs, design and development time can take months even up to years.

In most radar systems digital signal processing (DSP) is used extensively. DSP is the study of signals in a digital representation and the processing methods of these signals. The main goal of DSP is to filter to measure real-time analog signals. An analog-to-digital converter (ADC) is used initially to transform analog signals used in radar communications into digital signals. Many types of filters and transforms are used in DSP. These functions are implemented in some type of Application Specific Integrated Circuit (ASIC). A general conventional design flow for ASICs is as follows:

1. Functional Specifications
2. Design Partitioning
3. RTL (RTL) Design & Simulation
4. Functional Verification
5. Synthesis for Area & Timing Optimizations

6. Placement & Routing
7. Chip Fabrication

This design flow is limited due to the length of time needed to make an ASIC. The RTL and Simulation process (Step #3) itself can take many weeks or months to complete, depending on the complexity of the design. The design flow is also limited by the high costs associated with it and the ASIC's limited flexibility. To solve this problem, a speedy and adaptable design flow will be proposed by placing pre-defined modular components into a library. This library will consist of highly customizeable and configurable codes of DSP functions that can target either ASICs or field programmable gate arrays (FPGA) to produce circuits to suit the intended applications. The development of this library will be a time consuming process in itself, but once the library is complete all a designer must do is pick and choose which components from the library he or she wants to use. The components will be configurable so there will be limitless design possibilities. Performing this work in-house will save the Department of Defense (DoD) from having to out-source to companies such as Boeing or Raytheon, who could charge millions of dollars to produce such a product.

1.1 *Specific Issue:*

DSP is an extremely important function in radar applications. The processing of digital data must be performed as fast as possible so the warfighter has the advantage in any combat situation. One such component in DSP is the Fast Fourier Transform (FFT). The FFT is an algorithm for converting a digital signal in the time domain to a signal in the frequency domain. One of the original uses of the FFT was to distinguish between nuclear explosions and natural seismic events. These two phenomena produce different frequency spectra. By converting the signals to the frequency domain a distinction between the two events could be seen. Aircraft have different radar signatures, so by using the FFT on the radar signals received the pilot can see the aircraft's location and speed. In this research a configurable FFT will be developed for the aforementioned library. In addition, this FFT will be dynamic; i.e. it will be able to calculate different length FFTs real-time. This implementation will be using a 90 nm technology library from the Taiwan Semiconductor Manufacturing Company (TSMC). Results from this library will be compared to those of the AMI 350 nm library from Oklahoma State University. The 90 nm technology will provide for faster speeds and lower power consumption compared to those of the 350 nm library.

1.2 Problem Statement:

The problem to be solved is the demonstration of a modular digital radar library by designing and characterizing one possible component. The FFT being designed will be both configurable and dynamic. The configurable parameters can be changed pre-synthesis and the dynamic parameters can be changed at run-time. To keep the chip size small and power consumption low, a minimal hardward approach will be used. This will result in a longer design time for each component in the library but will allow for the most efficient design.

1.3 Scope and Assumptions:

It is assumed that readers of this paper will have a basic understanding of digital signal processessing and more specifically FFTs. Additionally, strong knowledge of the Very-High-Speed Integrated Circuit(VHSIC) Hardware Descriptor Language (VHDL) is required to understand the coding of the design. The software used for this research includes Modelsim for circuit simulations, MATLAB for simulations and error analysis, and the Cadence Encounter RTL compiler for synthesis and power, timing, and area analysis. A knowledge of simple digital logic components is also assumed. Such components include muxes, adders, and pipeline registers.

1.4 Thesis Organization:

The next chapter of this research project will discuss background information necessary to understand the scope of this project. A discussion of the mathematics and algorithms for the FFT is included. Additionally several current (within the past 5 years) FFT implementations will be analyzed and their results discussed. Chapter III will consist of the theory involved in the design of the FFT architecture and the methods of testing used. The results of the implementation and characterization will be discussed in chapter IV. Finally, a review and a look at future topics will be discussed in chapter V. All VHDL code will be viewable in the appendices.

1.5 Chapter Summary:

The purpose of this research project is to characterize and implement an FFT component for use at the Air Force Research Lab (AFRL). Pending a successful demonstration of this component, the component will be included in a future library of many configurable DSP functions, saving the U.S. military millions of dollars in addition to many months of design time.

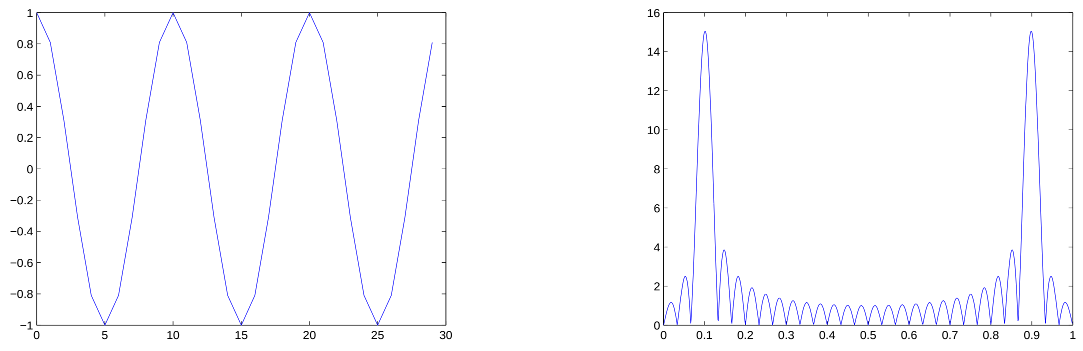
II. Background

This chapter provides an overview of the research involved in understanding the scope of this thesis. Fourier transforms and specifically FFTs are reviewed. Two FFT algorithms are discussed, as one will be used in the FFT implementation. Many FFT implementations are available in the IEEE database. Several of the most recent implementations and their claimed results are analyzed.

2.1 Fourier Transform

To understand the derivation and need for the Fast Fourier Transform, we will first look at the Fourier Transform. The Fourier transform and series are named after the French scientist and mathematician Joseph Fourier. The equation for the Fourier Transform is given in (2.1). It is a generalization of the complex Fourier Series [4]. This equation takes a signal in the time domain and transforms it into the frequency domain. Information such as frequency range and energy can be obtained from the frequency domain representation. Figure 2.1 shows an example of such a transformation.

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt \tag{2.1}$$



(a) (b)

Figure 2.1: Signal representation in (a) time domain (b) frequency domain

2.2 Discrete Fourier Transforms

The Fourier Transform worked on continuous signals. In fields such as signal processing, signals are usually sampled. These sampled signals are called discrete signals. To calculate the Fourier Transform of a discrete signal, the Discrete Fourier Transform (DFT) is used. James Tsui explains the two limitations of the continuous Fourier transform in his book [21]:

> First, the function in the time domain must be representable in closed form so that the Fourier integral can be performed. Thus, unless the input function can be written in closed form, it is impossible to evaluate the integral. Second, even if the time function can be written in closed form, it might also be difficult to find a closed-form solution to the integral.

The data to be transformed comes from an ADC, so it is digitized and the function in the time domain is unknown. Unlike the Fourier transform, the DFT can be performed on any kind of input data; therefore, its usage is unlimited [21]. Also, the results from a DFT are an approximate solution.

The general definition of the DFT is as follows: let $x(n), n = 0, 1, 2...., N-1$, be an N-point sequence. From [18], the definition of its discrete Fourier transform is

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j\frac{2\pi}{N}nk}, k = 0, 1, 2, ..., N-1 \tag{2.2}$$

For convenience, denote $e^{-j\frac{2\pi}{N}nk}$ by W_N, so equation (2.2) becomes:

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn}, k = 0, 1, 2, ..., N-1 \tag{2.3}$$

which can be expanded into

$$X(k) = x(0)W_N^0 + x(1)W_N^k + x(2)W_n^{2k} + ... + x(N-1)W_N^{(N-1)k} \tag{2.4}$$

The W_N term is called the n^{th} root of unity, also known as a "twiddle factor". This term was coined by Gentleman and Sande in 1966, and has since become widespread in the world of FFTs [9]. From equation (2.4), the calculation of each $X(k)$ requires N complex multiplications and N complex additions. Since $X(k)$ is calculated from 0 to N-1, the direct computation of the DFT requires on the order of N^2 multiplications and N^2 additions. The complexity of this equation is $O(N^2)$. For example, a 1024 point DFT would require approximately 2,097,152 operations! Fortunately there is an algorithm which will reduce the complexity from $O(N^2)$ to $O(N \log_2 N)$. For the same 1024 point FFT, only about 20,480 operations will be needed, a large decrease which means a faster calculation. This algorithm is called the Fast Fourier Transform.

2.3 Fast Fourier Transforms

In 1805, Carl Friedrich Gauss describes the critical factorization steps for the FFT. Almost 150 years later in 1965 James Cooley and John Tukey formally publish the algorithm for the FFT [7]. They exploited the symmetrical properties of complex exponentiation reducing the complexity to $N \log_2 N$. There are two variations of the FFT algorithm, the Decimation-In-Time (DIT) FFT algorithm and the Decimation-In-Frequency (DIF) FFT algorithm.

2.3.1 Decimation-In-Time FFT. Cooley and Tukey, using the Danielson-Lanczos Lemma from 1942 [16], developed what is known as the decimation-in-time FFT algorithm. This lemma is only applicable if the length is a power of 2. From now on, we will assume N, the length of the transform, is a power of 2. The allowable lengths in the VHDL implementation range from 4, 8, 16, ..., up to 1024. For the DIT algorithm, $x(n)$ is divided into two sequences, each of length $N/2$. The even-indexed samples and odd-indexed samples are grouped separately. Equation (2.2) can be rewritten as

$$\begin{aligned}
X(k) &= \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi nk}{N}} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-j\frac{2\pi(2n)k}{N}} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-j\frac{2\pi(2n+1)k}{N}} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-j\frac{2\pi nk}{\frac{N}{2}}} + e^{-j\frac{2\pi k}{N}} \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-j\frac{2\pi nk}{\frac{N}{2}}} \qquad (2.5) \\
&= DFT_{\frac{N}{2}}[[x(0), x(2), ..., x(N-2)]] + W_N^k DFT_{\frac{N}{2}}[[x(1), x(3), ..., x(N-1)]]
\end{aligned}$$

The simplifications in equation (2.5) show that all frequency outputs $X(k)$ can be computed as the sum of the outputs of two length $\frac{N}{2}$ DFTs, using the even-indexed and odd-indexed discrete samples respectively. The odd-indexed short DFT is multiplied by a "twiddle factor" term, W_N. Because the samples are split into two separate groups, this algorithm is called a "radix-2" algorithm. Other such algorithms exist for radix-4 and radix-8, but will not be discussed in this paper. Since the time samples are rearranged in alternating groups, this algorithm is called decimation in time. Figure 2.2 shows how this process begins by breaking the inputs up into two $N/2$ DFTs. The recombine stage shown in the figure is used to combine the samples in the correct order. This process is covered later. Now, the two $N/2$ stages can be broken down into four $N/4$-point DFS, as shown in Figure 2.3. This process is repeated until a series of two-point DFTs are reached. Figure 2.4 shows the flow graph for a two-point FFT. This structure is also known as a butterfly.

Figure 2.5 shows an example for a length of 8. Notice the "out-of-order" ordering of the inputs. Actually, this is bit-reversed ordering, and is a natural process due to the mathematics of the FFT. To obtain a bit reversed number simply take the binary equivalent, reverse the order of the bits, and recalculate the decimal equivalent from that. Table 2.1 shows how the numbers are bit-reversed.

This process also allows for in-place computation, which means the results of

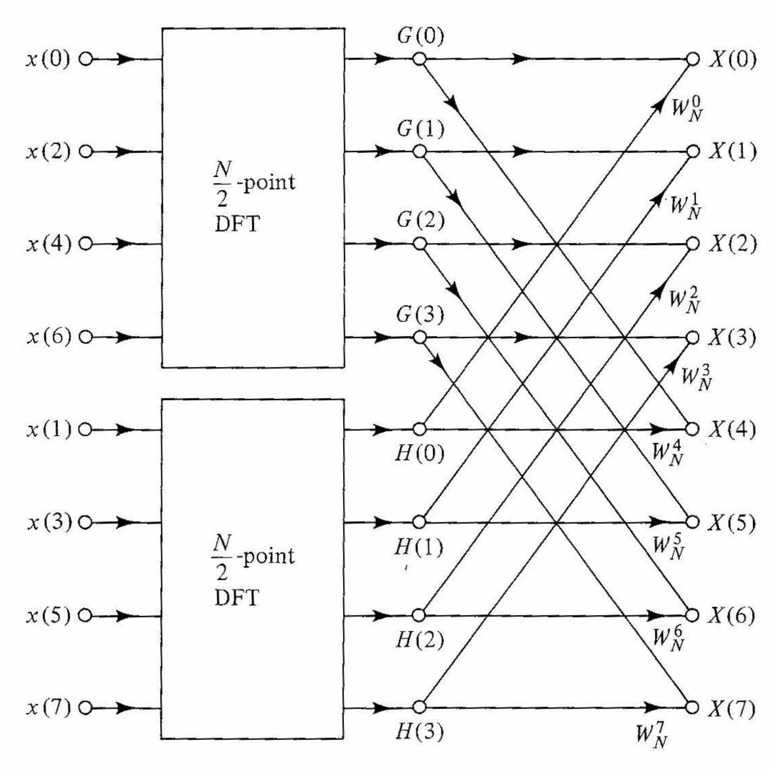


Figure 2.2: Decimation-in-time of a length N DFT into two length $N/2$ DFTs followed by a recombining stage. [18]

the calculations at any stage can be stored in the same memory locations as those of the input to that stage. This idea is illustrated in Figure 2.5. The calculations of $X(0)$ and $X(4)$ require the same two inputs. Once this calculation is complete the two inputs are no longer needed, so the calculated butterfly values of $X(0)$ and $X(4)$ can be stored in the memory locations of $X(0)$ and $X(4)$. Because of this, only $2N$ storage locations are needed.

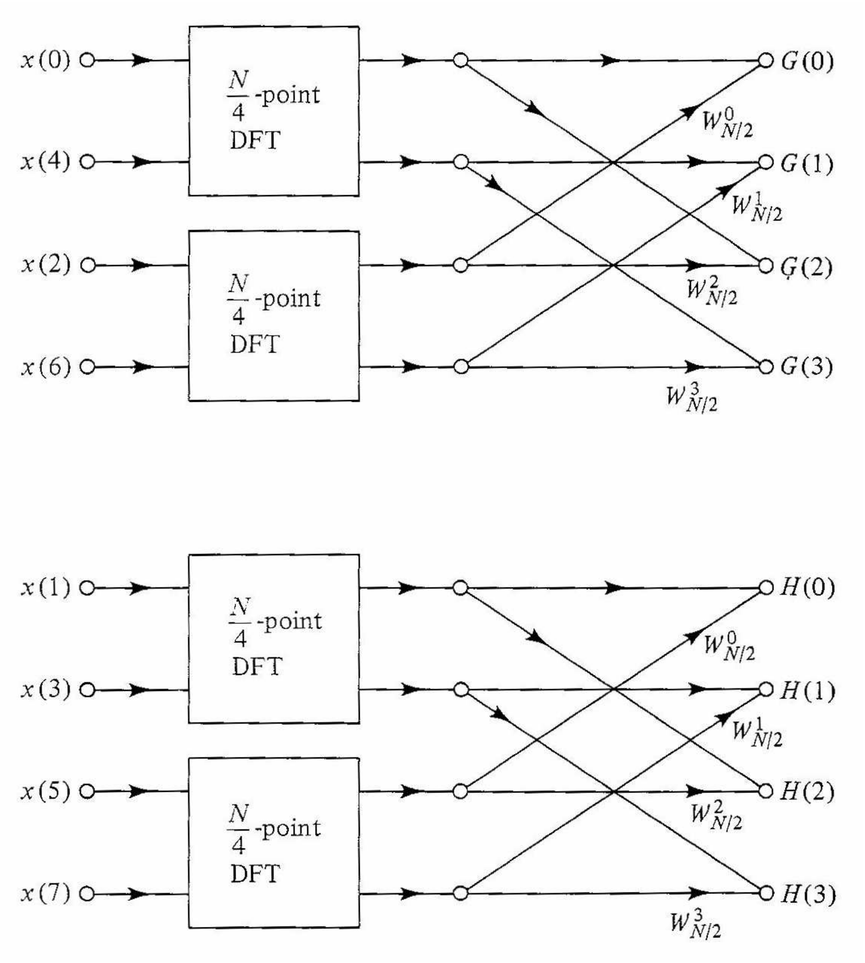


Figure 2.3: Decimation-in-time of a length N DFT into four length $N/4$ DFTs followed by a recombining stage. [18]

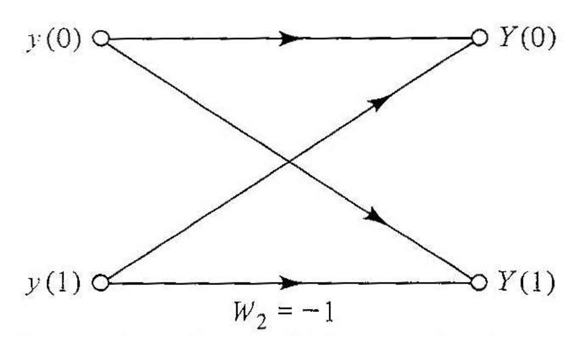


Figure 2.4: Flow graph for computation of a two-point DFT. [18]

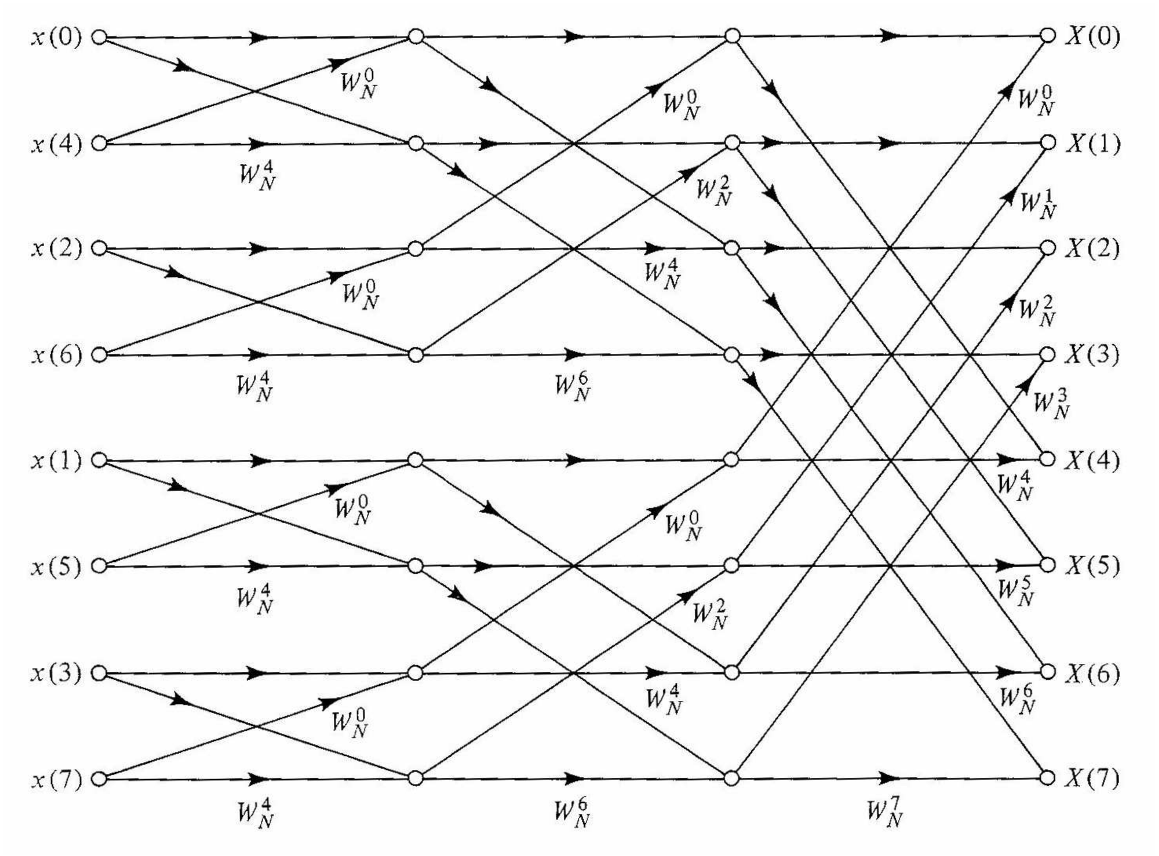


Figure 2.5: Decimation-in-time of a length 8 DFT. [18]

Table 2.1: Bit-reversed order for N=8.

Decimal Number	Binary Representation	Bit-Reversed Representation	Decimal Equivalent
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

2.3.2 Decimation-In-Frequency FFT. Two gentlemen by the name of Sande and Tukey developed the decimation-in-frequency algorithm [19]. The DIF algorithm works backward from the DIT algorithm. Instead of dividing the input sequence $x(n)$ into smaller subsequences, the output sequence $X(k)$ is subdivided. The algorithm consists of arranging the DFT into two parts: calculation of the even-numbered frequency indices $X(k)$ for $k = 0, 2, 4, ..., N-2$ and calculation of the odd-numbered frequency indices $k = 1, 3, 5, ..., N-1$, or $X(2r)$ and $X(2r+1)$, respectively. We have

$$\begin{aligned} X(2r) &= \sum_{n=0}^{N-1} x(n) W_N^{2rn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(n) W_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x(n+\frac{N}{2}) W_N^{2r(n+\frac{N}{2})} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(n) W_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x(n+\frac{N}{2}) W_N^{2rn} 1 \\ &= \sum_{n=0}^{\frac{N}{2}-1} (x(n) + x(n+\frac{N}{2})) W_{\frac{N}{2}}^{rn} \\ &= DFT_{\frac{N}{2}}(x(n) + x(n+\frac{N}{2})) \end{aligned} \tag{2.6}$$

and

$$\begin{aligned} X(2r+1) &= \sum_{n=0}^{N-1} x(n) W_N^{(2r+1)n} \\ &= \sum_{n=0}^{\frac{N}{2}-1} (x(n) + W_N^{\frac{N}{2}} x(n+\frac{N}{2})) W_N^{(2r+1)n} \\ &= \sum_{n=0}^{\frac{N}{2}-1} ((x(n) - x(n+\frac{N}{2})) W_N^{n}) W_{\frac{N}{2}}^{rn} \\ &= DFT_{\frac{N}{2}}(x(n) - x(n+\frac{N}{2}) W_N^{n}) \end{aligned} \tag{2.7}$$

Notice only the odd-indexed frequencies are multiplied by the twiddle factors. Also the frequency samples are computed separately in alternating groups, hence the decimation in frequency designation. The inputs of the DIF FFT are in order and the outputs are now in bit-reversed order, opposite of the DIT algorithm. It is for this reason the DIF algorithm is chosen for the VHDL implementation. Either way, re-ordering hardware is necessary to arrange the data before or after the FFT calculation. Figure 2.6 shows the first stage with the FFT being split into two $N/2$ DFTs. These $N/2$ DFTs are broken down until a length-two DFT is found. This is shown in Figure 2.7.

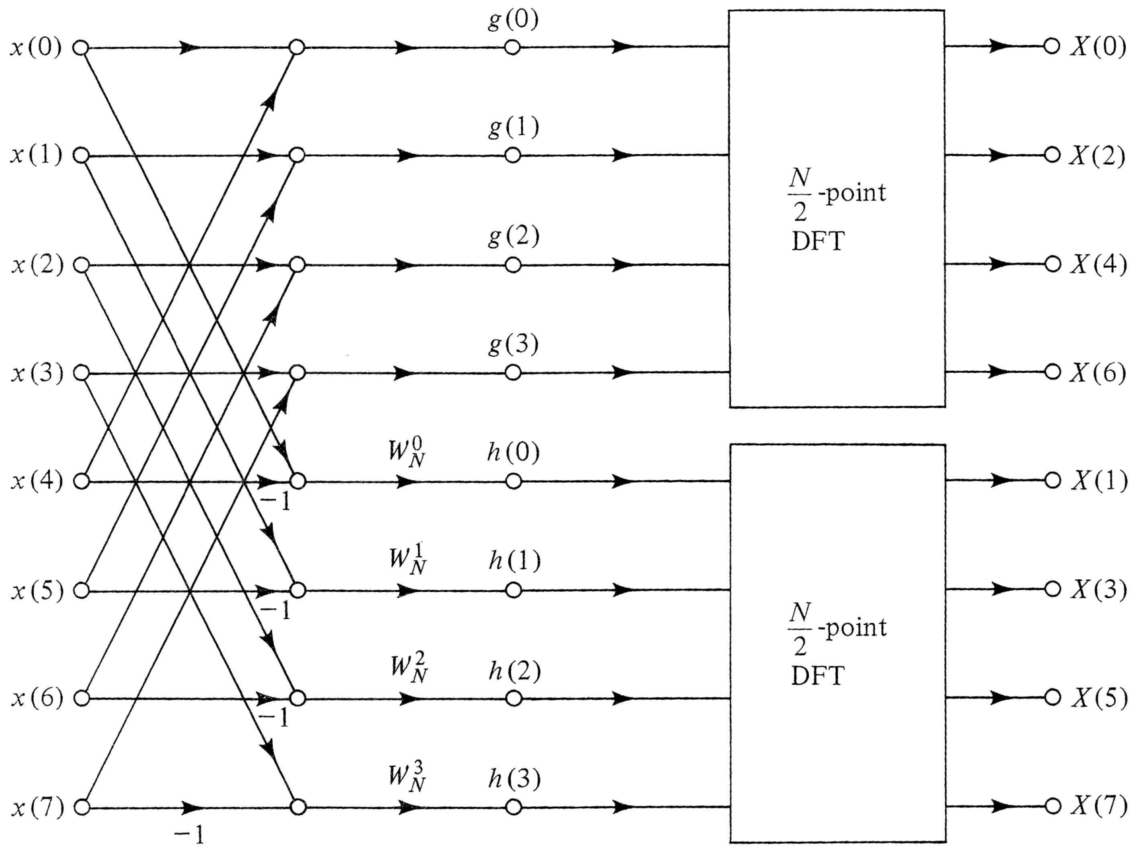


Figure 2.6: DIF of a length N DFT into two length $N/2$ DFTs. [18]

2.4 VHDL

With the background and mathematics for a FFT in place, a vehicle to create the FFT circuit will now be discussed. There are several high-level languages which can be used to describe a digital circuit. VHDL is a popular design entry language for

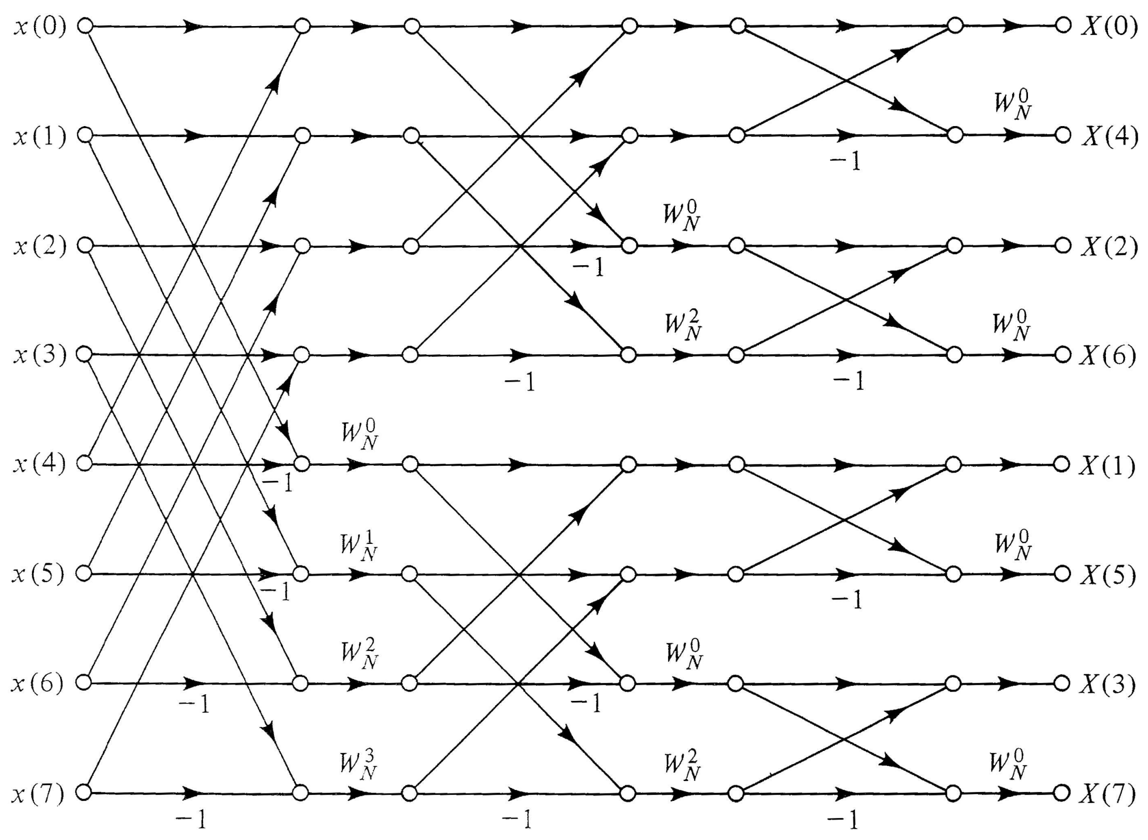


Figure 2.7: DIF of a length 8 DFT. [18]

FPGAs and ASICs. Another popular language is Verilog. One advantage VHDL has over Verilog is the ability to use *generate* statements. *Generate* statements are used to include many concurrent VHDL statements. In a modular design, *generate* statements will be used heavily to create the module using the least amount of transistors, thus reducing power, timing, and cell area. Once the VHDL code has been tested for errors and the simulations are correct, the next step is synthesis. Synthesizing takes the high-level description and produces a gate netlist. The gate netlist is generated by the Cadence Encounter RTL Compiler software and uses cells from the TSMC 90 nm library.

2.5 Other FFT Implementations

By looking at other FFT implementations one can get an understanding of what technologies were used, what the targeted results are, and any other novel ideas in an FFT design. Performing a search in the IEEE Xplore online database, the Opencores website (www.opencores.org), or Google on FFTs will show many different implementations of FFTs in VHDL and/or Verilog. In [23] a reconfigurable FFT which can compute lengths from 4 to 1024 is discussed. The author uses a radix-2 FFT algorithm. An overall view of the architecture is shown in Figure 2.8.

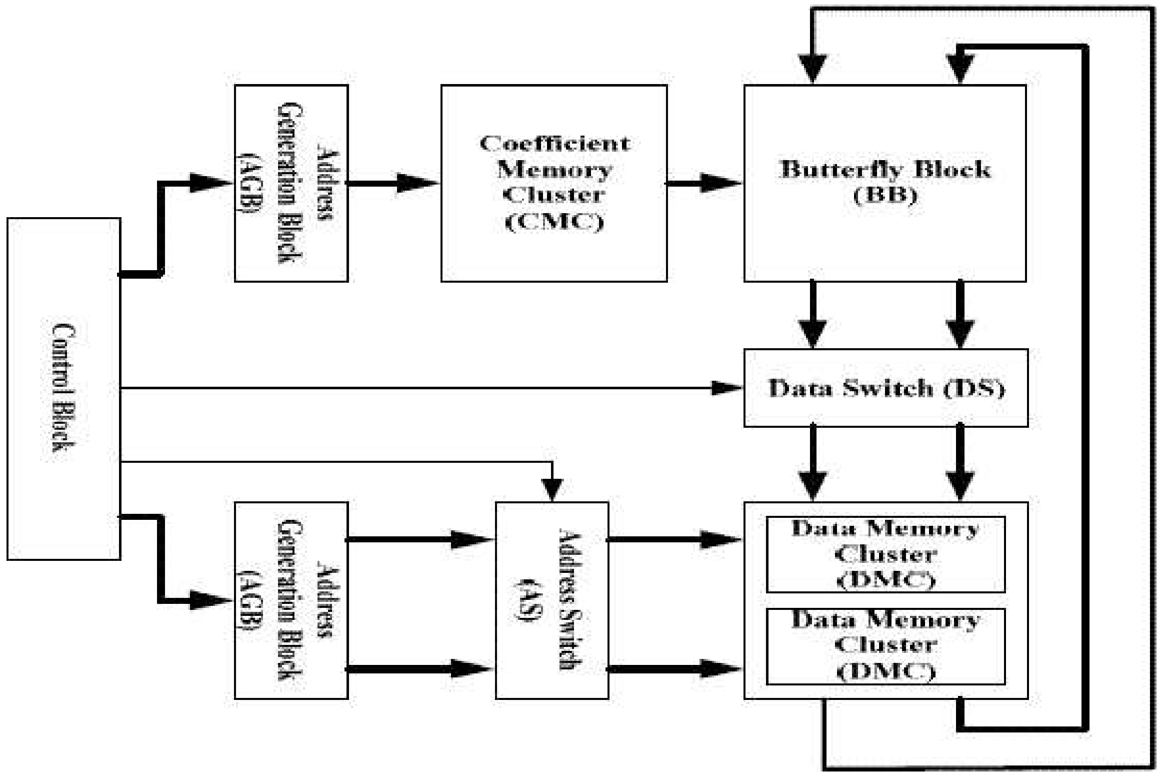

Control Block
Address Generation Block (AGB)
Coefficient Memory Cluster (CMC)
Butterfly Block (BB)
Data Switch (DS)
Address Generation Block (AGB)
Address Switch (AS)
Data Memory Cluster (DMC)
Data Memory Cluster (DMC)

Figure 2.8: Overall architecture of reconfigurable FFT processor [23]

The butterfly block (BB) carries out the butterfly calculations. Twiddle factors are stored in memory, called the coefficient memory cluster (CMC). The module stores 512 coefficients, enough to satisfy the requirements of a length 1024 FFT. The 512 twiddle factors are divided into 64 smaller modules called coefficient memory modules (CMM), with each module storing 8 values. Different coefficient sets are obtained by combining various CMMs. One set of CMMs will provide twiddle factors for a length 16 FFT. For larger lengths, CMMs are combined together to form a larger memory. The DMC, or data memory cluster, is composed of two 512x32-bit memories, giving a total of 1024 memory locations. The address generation block (AGB) generates addresses for both the DMC and CMC. These reconfigurable modules can compute addresses for different FFT lengths. The data switch (DS) routes the butterfly calculations to the correct DMC modules, using the addresses which are determined by the address switch (AS). The CB, or control block, contains counters which generate addresses and timing for the entire design. The author used the Verilog language to design the processor, and the design was synthesized to the UMC 0.18μm CMOS standard cell library with the Synopsys Design Compiler [23]. Table 2.2 shows the power and area results after synthesis. The area is constant because this processor is able to compute FFTs of length 16 through 1024. With each increase in FFT length, the consumed power increases. This is expected because more calculations are performed with larger length FFTs.

Table 2.2: Power and Area Results [23]

FFT Size	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
Power Consumption (mw)	4.7	7.9	8.3	13.0	26.1	49.7	81.6
Area (mm2)	2.9						

In articles [10], [11], and [20] a pipelinable FFT architecture is presented. This type of architecture will ultimately be used in the design of the configurable/dynamic FFT proposed in this research. As such, the overall design will not be discussed until a later chapter. Pipelining the FFT processor allows for faster speeds to be achieved. In [11] the author designs a length 1024 FFT in VHDL, and synthesizes with a 0.5μm Complementary Metal-Oxide Semiconductor (CMOS) technology. Speeds of about 20 MHz have been achieved. This design is not configurable and can only calculate a length 1024 FFT. Another design following the pipeline architecture is mentioned in [20]. Again, this implementation is limited to a length of 1024. This author uses Handel-C to implement the FFT processor. Handel-C is a direct C-to-hardware language, and can be synthesized directly to high density FPGA devices from Altera or Xilinx [1]. It is based on the C programming language. The author reports a speed of 82 MHz for a 1024-point FFT.

2.6 Chapter Summary

The background of the FFT was discussed, along with several algorithms used to compute FFTs. Several current FFT implementations and their results were also briefly examined. These results will be compared to the results of this research.

III. Methodology

The methodology for the dynamic configurable FFT will now be discussed. The goal of this research is to show the feasibility of creating a library with many configurable DSP modules. The design to be implemented and demonstrated is a configurable dynamic FFT. The design for the FFT architecture is based on [10], [11], and [20]. Information on twiddle factors is found in [3]. The overall design is discussed first, followed by a detailed analysis of the major components found in the architecture. Minor components such as muxes and shift registers are assumed to be known, so their design will be omitted. One of the main reasons for creating configurable components is to be able to take a generic component and conform it to a specific application. These configurable parameters are processed before synthesizing. The configureability of the FFT is shown in Table 3.1.

Table 3.1: Configurable parameters.

Parameter	Description
input_width	bit width of real and imaginary parts of input data
output_width	bit width of real and imaginary parts of output data
tf_width	bit width of real and imaginary parts of twiddle factors
log2N	maximum length of FFT (log_2 of length); integer between 2 and 10
r1-r10	radix position of fixed rounders
c1-c10	clipping required for each fixed rounder
ri	input register; none or plr (pipeline register)
ro	output register; none or plr (pipeline register)
pl	pipeline FFT; yes or no
rt	register reset type; none, synch, or asynch
nc	number of different lengths; integer between 1 and 8

3.1 Overall Design

The inputs and outputs for the FFT are shown in Table 3.2. The FFT implementations is based on the Radix-2^2 Single-path Delay Feedback (R2^2SDF) architecture, and uses the DIF algorithm. Input data is processed in-order and the output is produced in bit-reversed order. The FFT receives $N/2^{size}$ complex inputs sequen-

Table 3.2: Inputs and Outputs.

Parameter	Description
d_in_r	real part of complex data input
d_in_i	imaginary part of complex data input
q_out_r	real part of complex data output
q_out_i	imaginary part of complex data output
frame_in	framing control signal; forces FFT calculation to begin on next input sample
frame_out	framing control signal; next output is result of new FFT calculation
clock	rising edge sensitive clock control signal
reset_n	active low control signal for reset
size	dynamic control signal selects length of current FFT; $length = \frac{N}{2^{size}}$

tially and the first output sample appears after $N/2^{size} - 1$ samples. The *size* signal controls the dynamic property of the FFT. Changing this input changes the current size of the FFT. An overview of the architecture is shown in Figure 3.1. The BF2I and BF2II are the butterfly modules. The BF2I is the typical module which was described earlier, and the BF2II is essentially the same except it takes into account the $-j$ twiddle factors and computes them automatically. The boxes above each butterfly module are shift registers, with the number inside the box describing how many shifts it performs. The $W1(n)$ through $W4(n)$ variables are the twiddle factors. These are multiplied with the output of the BF2II module and passed to the next BF2I module. The twiddle factors are approximated and stored in a ROM. Each butterfly module has control signals which determine the calculation the module performs. These signals are generated from a timing controller. In addition to the control signals, the timing controller generates the addresses for the twiddle factor ROMs. The length of the FFT in Figure 3.1 is 256. If one wanted to compute a 128 length FFT, the same architecture would be used. The differences would be the shift registers would be halved (i.e. 128 stage shift register becomes a 64 stage shift register) and the final BF2II module would be bypassed.

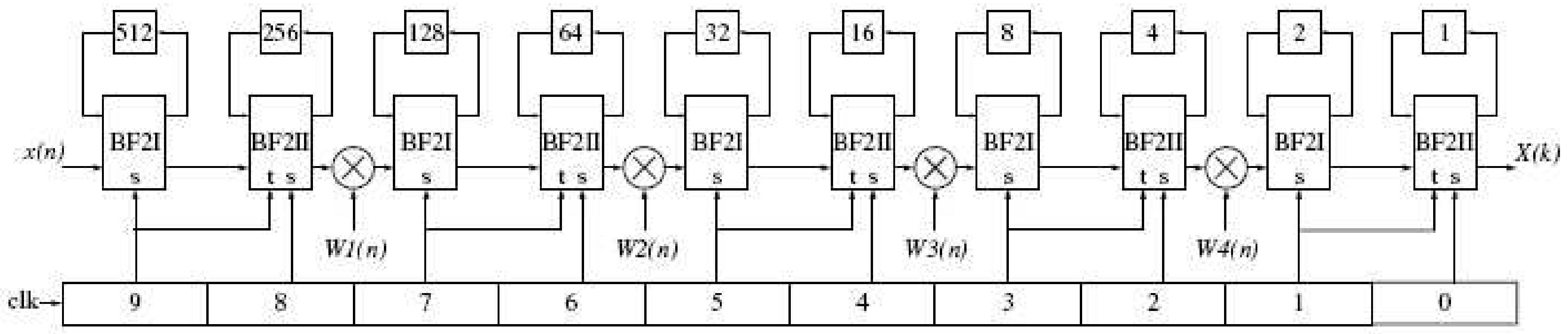


Figure 3.1: R2^2SDF FFT Architecture for N=1024 [20]

3.2 Butterfly Modules

Two butterfly modules are used, a *BF2I* and *BF2II*. On the first $N/2$ cycles, the 2-to-1 multiplexors in the first butterfly module (*BF2I*) are set to '0' and the module is idle. The input data is shifted into the shift registers until they are filled. On the next $N/2$ cycles, the butterfly module computes an N/2-point DFT with the input data and the data stored in the shift registers. The following equations describe the operations:

$$\begin{aligned} Z(n) &= x(n) && 0 \leq n < N/2 \\ Z(n+N/2) &= x(n+N/2) && 0 \leq n < N/2 \\ Z(n) &= x(n) + x(n+N/2) && N/2 \leq n < N \\ Z(n+N/2) &= x(n) - x(n+N/2) && N/2 \leq n < N \qquad (3.1) \end{aligned}$$

A physical implementation of the *BF2I* module is shown in Figure 3.2

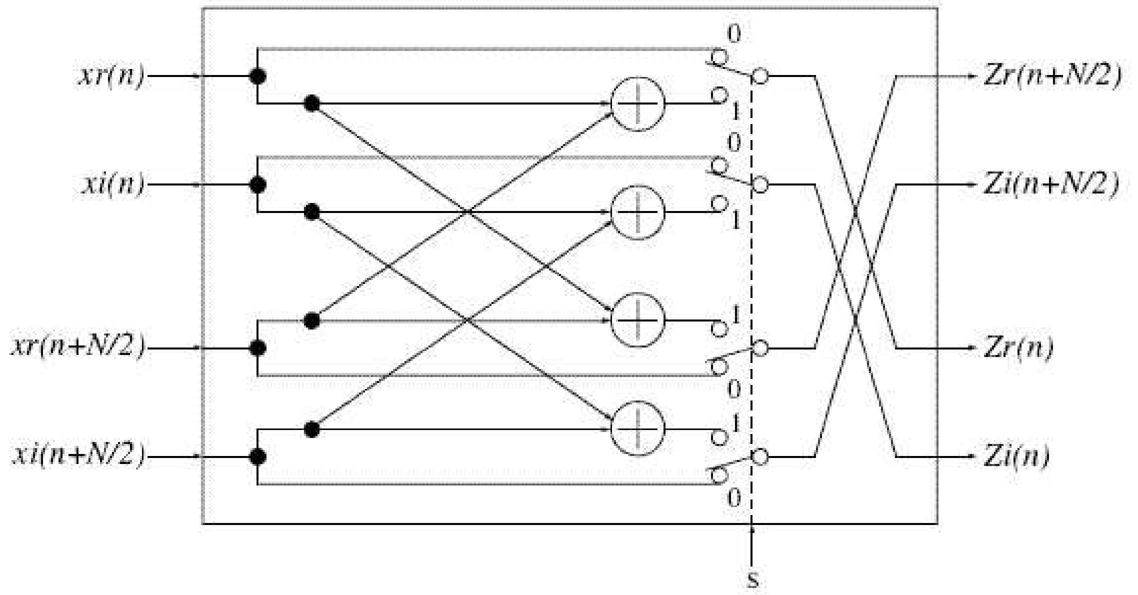


Figure 3.2: *BF2I* Module [3]

The *BF2II* module is similar in operation, except it takes into account the trivial twiddle factor multiplication of $-j$. The following equations describe the operations:

$$
\begin{array}{rcl}
Z(n) = x(n) & & 0 \leq n < M/4 \\
Z(n+N/2) = x(n+N/2) & & \\
\hline
Z_{real}(n) = x_{real}(n) + x_{imag}(n+N/2) & & N/4 \leq n < N/2 \\
Z_{imag}(n) = x_{imag}(n) - x_{real}(n+N/2) & & \\
Z_{real}(n+N/2) = x_{real}(n) - x_{imag}(n+N/2) & & \\
Z_{imag}(n+N/2) = x_{imag}(n) + x_{real}(n+N/2) & & \\
\hline
Z(n) = x(n) & & N/2 \leq n < 3N/4 \\
Z(n+N/2) = x(n+N/2) & & \\
\hline
Z(n) = x(n) + x(n+N/2) & & 3N/4 \leq n < M \\
Z(n+N/2) = x(n) - x(n+N/2) & &
\end{array} \qquad (3.2)
$$

A physical implementation of the *BF2II* module is shown in Figure 3.3

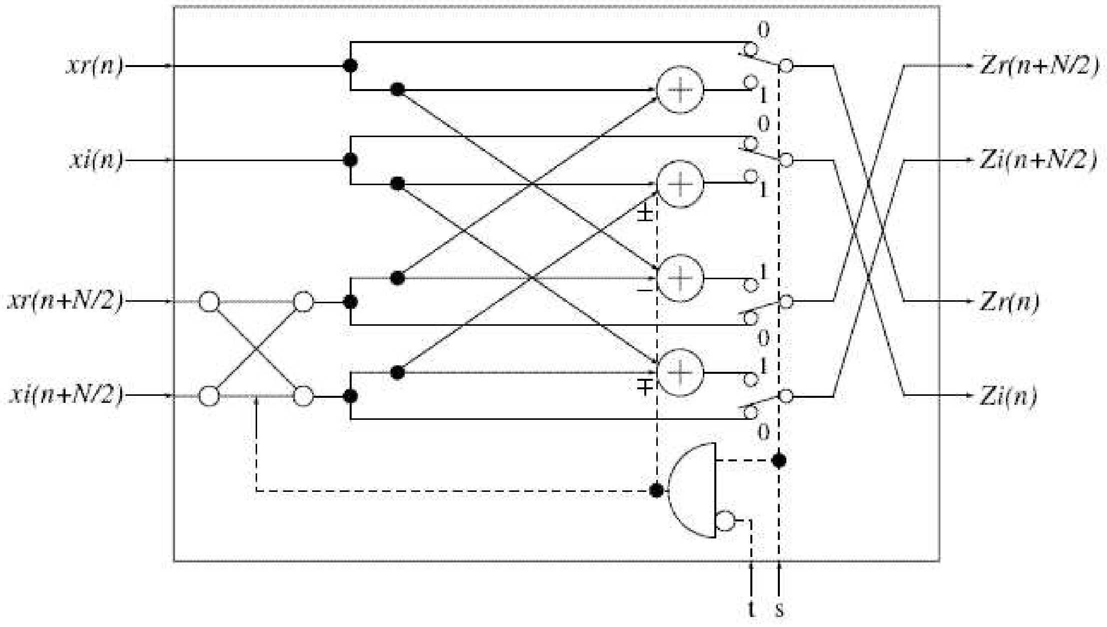


Figure 3.3: *BF2II* Module [3]

3.3 Twiddle Factors

The generation of the twiddle factors, or complex roots of unity, for the FFT will determine the amount of error between the VHDL implementation and real FFT calculation in MATLAB. These values are precomputed and their binary representation is stored in a ROM. The bit width selected for the twiddle factors will control the amount of error. Choosing a large bit width guarantees greater accuracy at a cost of larger die area and higher power consumption. On the other hand, a small bit width generates a smaller area and power consumption but larger error. In the VHDL implementation, they are based on the module length M and the FFT length N. The equation to calculate these values is

$$W_p(n) = e^{\frac{-j2\pi q(n)}{N}}, 0 \leq p \leq \log_4(N) - 2 \tag{3.3}$$

expanding this into a trigonometric expression yields

$$W_p(n) = cos(\frac{2\pi q(n)}{N}) - jsin(\frac{2\pi q(n)}{N}), 0 \leq p \leq \log_4(N) - 2 \tag{3.4}$$

where

$$\begin{aligned} q(n) &= 0 * 4^p * n \\ &= 2 * 4^p * (n - \frac{M_p}{4}) \\ &= 1 * 4^p * (n - \frac{M_p}{2}) \\ &= 3 * 4^p * (n - \frac{3M_p}{4}) \end{aligned} \tag{3.5}$$

The calculated twiddle factors range between 1 and -1. They are now scaled between the max and min values based on the twiddle factor bit width. For example, if the bit width is 10, the twiddle factor would be scaled to a value between 511 ($2^9 - 1$) to

Table 3.3: *Wx* Modules.

Wx	Contains twiddle factors for lengths...
0	1024, 512
1	1024, 512, 256, 128
2	1024, 512, 256, 128, 64, 32
3	1024, 512, 256, 128, 64, 32, 16, 8

-512 (-2^9). The real and imaginary calculations are summarized as:

$$Real = round(cos(\theta) * (2^{TwiddleFactorBitWidth-1} - 1)) \tag{3.6}$$

$$Imag = round(sin(\theta) * (2^{TwiddleFactorBitWidth-1} - 1)) \tag{3.7}$$

Upon storing the twiddle factors in the ROM, they must be offset by $3M/4$ samples to ensure they are aligned with the first sample in the block. If the FFT is configured for dynamic sizes, all possible twiddle factor ROMs must be made available. For example, if N=16 and nc=2, FFTs of length 16 and 8 can be calculated. The twiddle factors for N=16 are different than those for N=8. In this case, the module *W3* contains both ROMs and a multiplexor is used to select the correct twiddle factors based on the currently selected size. This is the case for all dynamic lengths between 1024 and 8. Note, length-4 FFTs do not incorporate twiddle factors. The *W3* module is configurable based on N and nc, so the minimal logic is created. Table 3.3 shows the *Wx* modules and the ROMs they may contain. Figure 3.4 shows the schematic for a W3 module for a 1024 length FFT with the number of dynamic choices equal to 8. The output of each ROM is passed to the output using the 8-to-1 mux with the size signal performing the selection. Smaller dynamic choices use smaller muxes to conserve chip space.

The twiddle factors are generated by running a simulation on the *ROMGenerate.vhd* file. The configurable parameters are specified in the generic listing. This

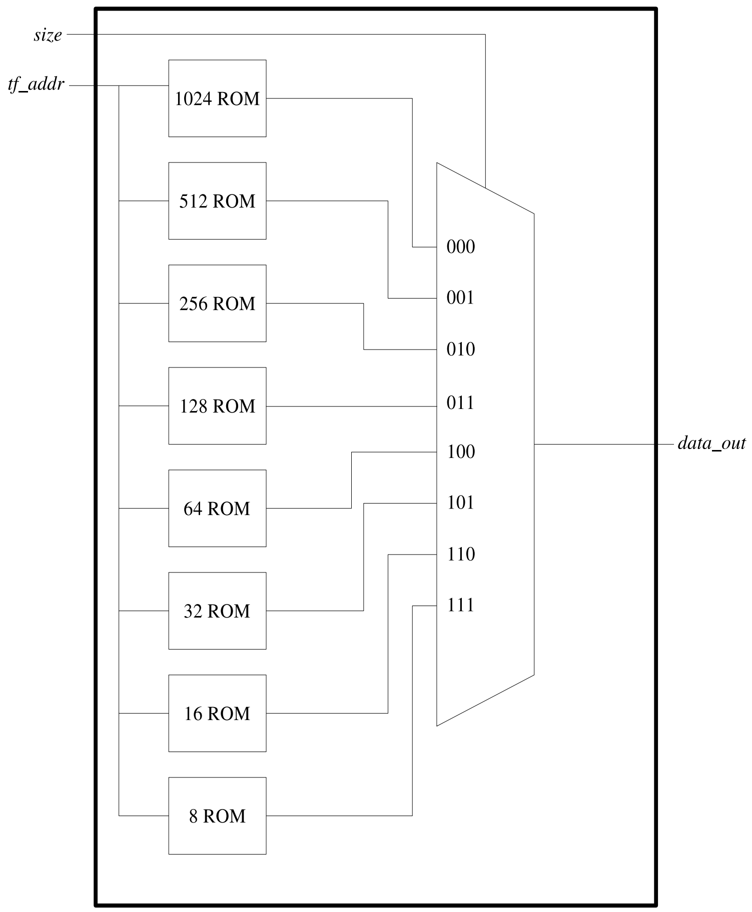


Figure 3.4: *W3* Module for a 1024 point dynamic FFT

code will generate the twiddle factors automatically and store them in a file called *TwiddleFactors.vhd*. This file will need to be included in order for the entire design to be elaborated and synthesized.

3.4 Timing

Timing is essential for all the components to work correctly. The timing controller is simply a log_2N-bit up counter. The timing signals are passed onto the butterfly modules and the twiddle factor ROMs. This component also generates the *frame_out* signal designating the completion of the FFT calculation and the first result will appear on the *q_out* line. The *frame_out* signal is generated when the counter "rolls over" to '0'. Due to the dynamic portion of the hardware, the value at which the counter rolls over depends on the length of the FFT and the value of the *size* signal. If a 1024 point FFT is calculated, a 10-bit counter is used. The rollover value would then be 1111111111. If the FFT is configured to be dynamic, the rollover value is shifted to the right by the integer value of the *size* signal. For example, for a 1024 point FFT, a size signal of 001 would shift the rollover values to the right by one producing a value of 0111111111 which corresponds to a 512 length FFT. Each stage can handle two different FFT lengths, but because the control signals to each stage are static, the timing controller will shift the count values left one bit based on the size signal. If pipelining is enabled, the control signals and twiddle factor addresses will also have to be pipelined. This is handled in a module called *TimingPLR*. Excluding the first stage, pipeline registers are placed around the multipliers in each stage if enabled. This placement of pipeline registers was chosen because the critical path, or longest delay in a circuit, always passes through the multipliers. Placing pipeline registers around the multipliers shortens the critical path, thus increasing the frequency at which the FFT can operate. The timing signals for each subsequent stage must then be delayed by two clock cycles to make sure they meet up with the correct values in the computation, hence the two pipeline stages in the *TimingPLR* component. Because the complex multipliers are sandwiched between pipeline registers, the twiddle factor address must be delayed by one cycle initially, then by two cycles for each subsequent stage. The *TimingPLR* module takes as an input the twiddle factor address signal, but has two outputs for the address. The first output is delayed by one cycle to be used in the current stage and the second is delayed by two cycles to be used in the next stages. Figure 3.5 shows the timing controller and first set of pipeline stages for a length-1024 FFT.

3.5 Stage 1

Figure 3.6(a) shows a diagram of the stage 1 component, and Figure 3.6(b) shows the corresponding flow graph. The dashed lines correspond to complex data. For this example a $N/4$ length such as 4, 16, 64, etc is assumed. Data enters the

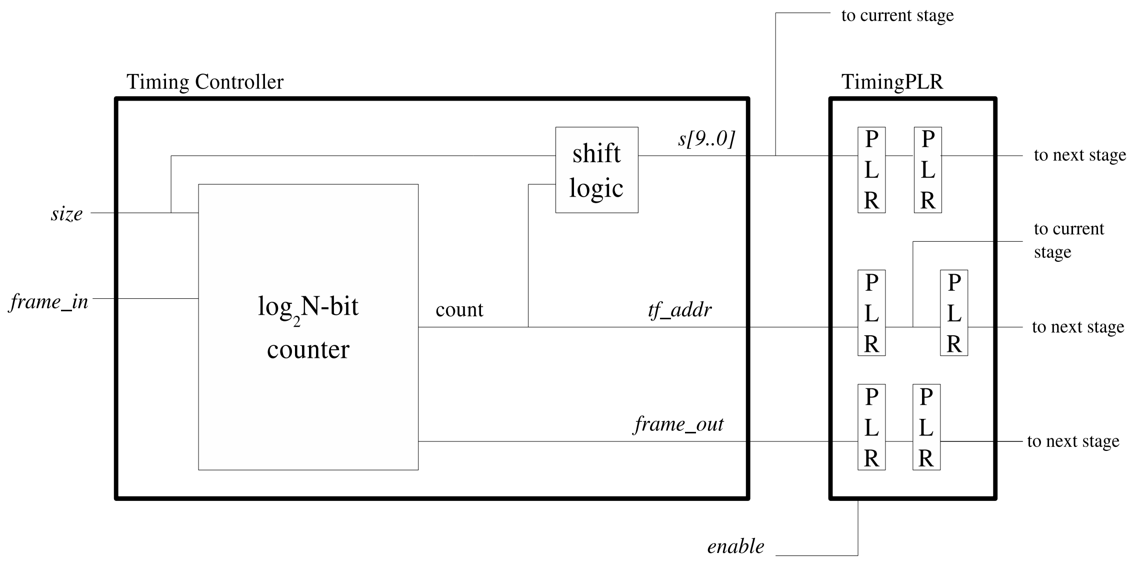


Figure 3.5: Timing Controller and pipeline for a 1024 point dynamic FFT

first butterfly module at the $x(n + M/2)$ input. The first two values are passed into the 2-stage shift register. The third and fourth data point are "butterflied" and the output is passed onto the fixed rounder. These represent the top two lines in the second stage shown in Figure 3.6(b). The outputs to go into the bottom two lines are put back into the shift register and held for two cycles until they can be placed into the second butterfly module. The $-j$ multiplication is build into the *BF2II* module. This module will compute the last two 2-point DFTs and pass the output to the final fixed rounder and *q_out* in bit-reversed order. If the length were an $N/2$ length such as 8, 32, 128, etc, then only *BF2I* would be used to compute a 2-point DFT and the *BF2II* would be bypassed. The sign extend extends the bit width by one because the *BF2I* and *BF2II* modules automatically increase the bit width by one during operation.

3.6 Stage X

Figure 3.7 shows a diagram of the rest of the stages. These generic stages are essentially the same except for the configurable parameters of the fixed rounder, the shift registers, and $W_x(n)$ modules. Operation is the same as in the first stage, but

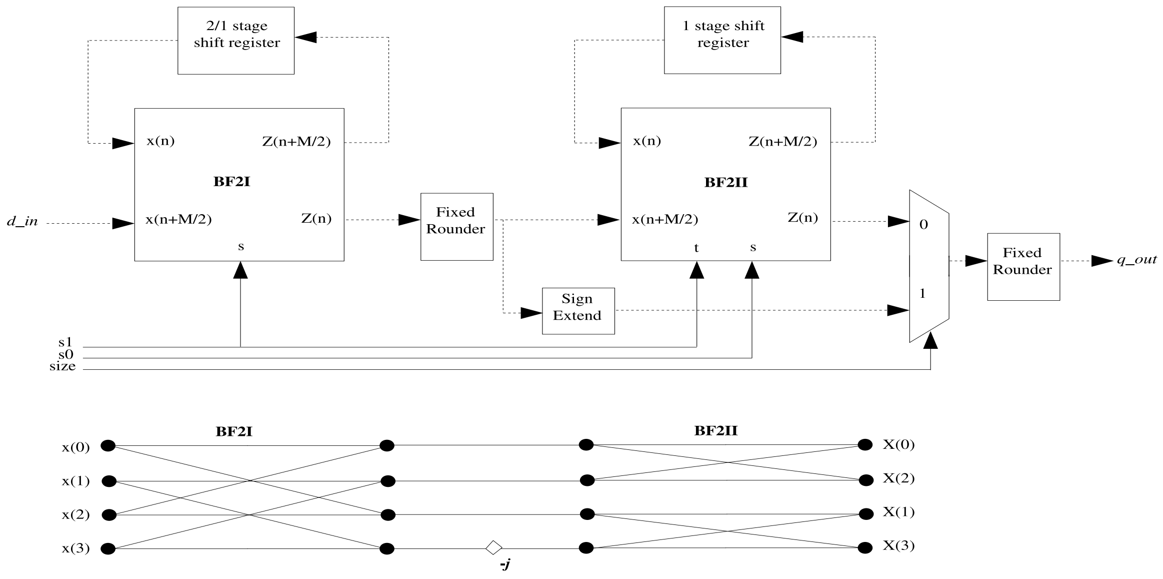


Figure 3.6: Stage 1 of FFT and Flow Graph Comparison

this time the twiddle factors and a complex multiplier are included. If the FFT is configured for pipelining, then pipeline registers are placed before and after the complex multiplier.

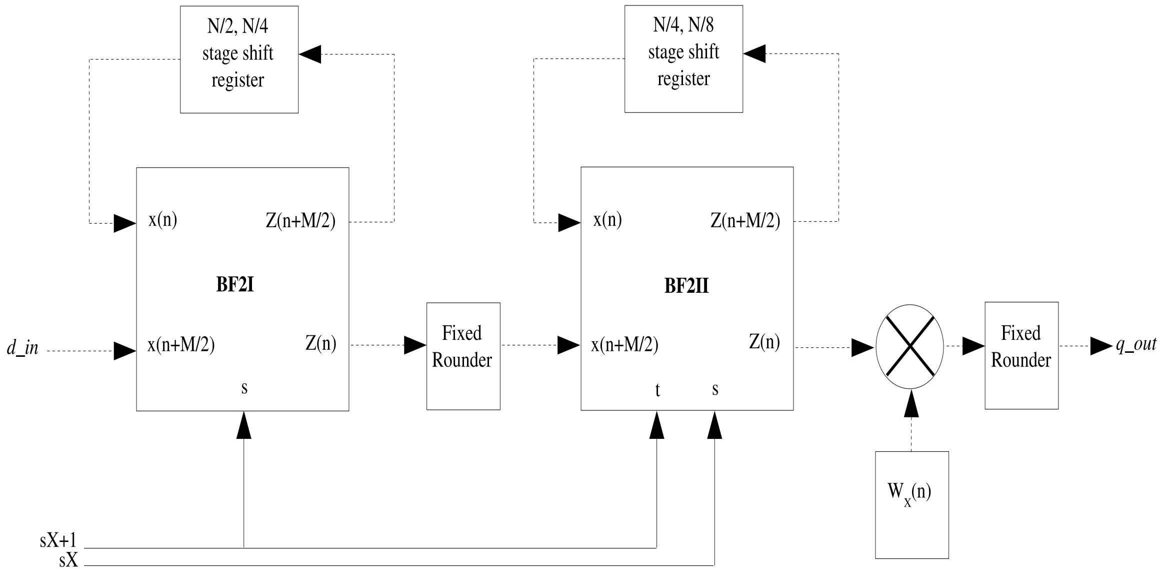


Figure 3.7: Stage X of FFT

3.7 Completed Design

Connecting the timing controller and the different stages togethers is not as easy as it sounds. Due to the configurable and dynamic nature of the FFT, all possible scenarios must be considered while keeping the hardware usage at a minimum. As an example, a 1024 length dynamic length (nc = 8) FFT is shown in Figure 3.8.

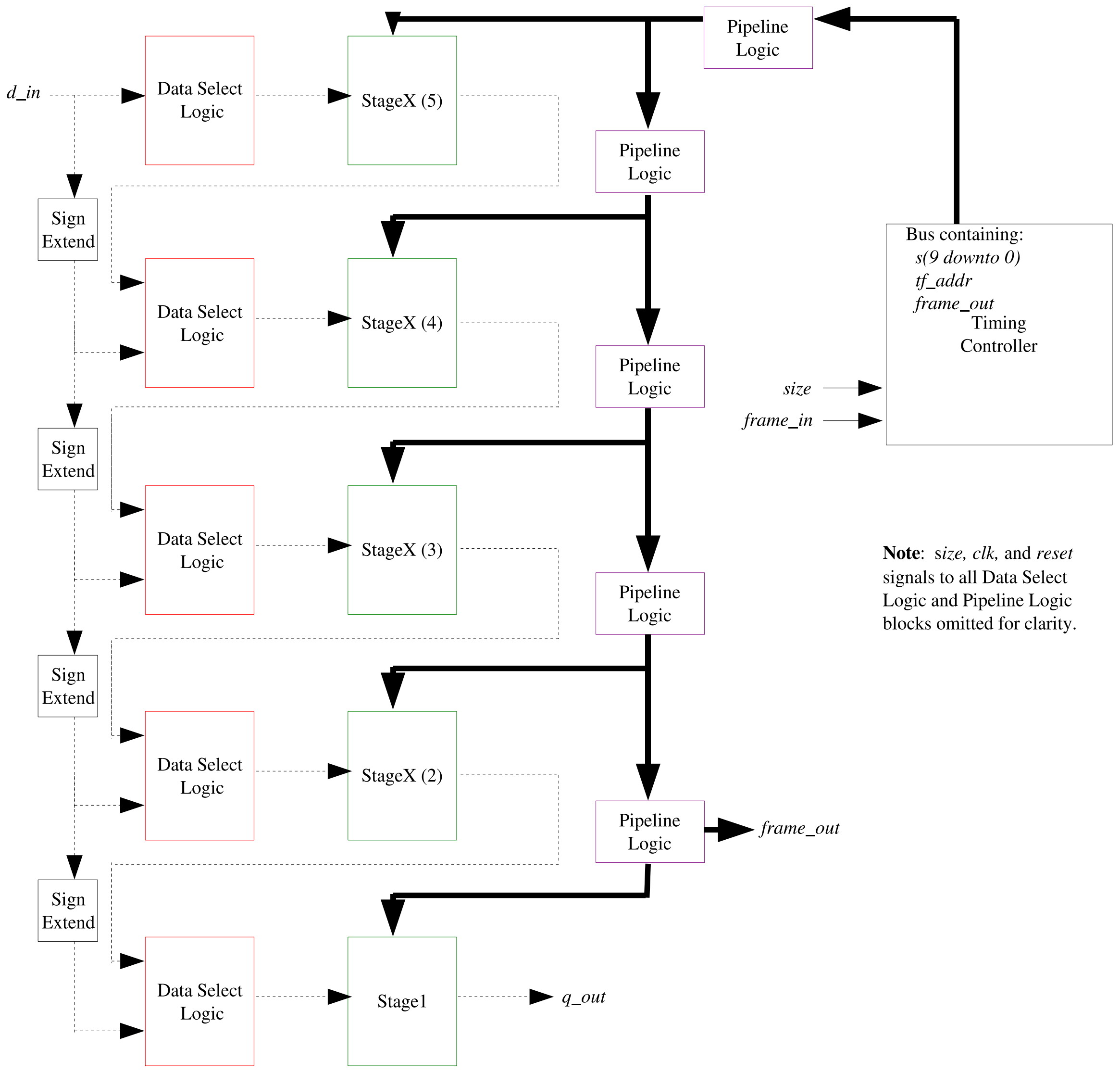


Figure 3.8: Layout of a 1024 length dynamic FFT

This configuration has data select logic to determine which stage the input data should begin. For example, for a 1024 or 512 length the input data should enter stage 5. For a length 256 or 128, the data should bypass stage 5 and begin in stage 4.

In addition, the data select logic selects between using the *d_in* data or the results from the previous stage. This logic is essentially a mux (2 or 4 input mux with the unused inputs grounded). The select bit(s) logic for the mux is generated using Karnaugh maps and the *size* signal. If a stage is not to be used, then the inputs are grounded to prevent usage of the elements within the stage, reducing power usage and heat generation. The sign extender between each stage extends the bit width of the input data, which is due to the bit growth of the butterfly modules and twiddle factor multiplications. With a static 1024 length FFT, there would be no data select logic blocks; instead the data would pass right through. *Generate* statements are used heavily for this portion of FFT code. The structural definition of the code begins by instantiating a timing controller. After that, each case is broken down by the configured length. In each of these cases, all possible lengths (based on the *nc* parameter) is broken down. Again, to prevent unnecessary chip space usage, *generate* statements are used. If pipelining is configured, then pipeline registers are placed to control the arrival time of the butterfly module control signals, twiddle factor addresses, and the *frame_out* signal. The logic for the pipeline controls whether the pipeline is turned on or not. In the 1024/512 case, all pipeline registers should be functioning. For the 256/128 case, the first set of pipeline registers should be turned off so the signals arrive correctly. The logic for controlling this functionality is again determined by Karnaugh maps.

3.8 Testing Procedure

There are two major areas on which testing will be performed. First, the Cadence Encounter RTL compiler will be used to analyze the timing, power, and chip area used by all the configurations of the FFT. For the timing analysis, all the possible lengths (4 to 1024) and all possible dynamic sizes (1 to 8) will be synthesized for both the pipelined and non-pipelined versions. Power and cell usage analysis will be performed with the same parameters, except only the pipelined version will be examined. This will give an overall outlook on the configurable properties and what effects it has. A PERL script, located in Appendix A, was written to perform these tests. The script first will modify the configurable parameters in the *FFT.vhd* file. Next, the Cadence RTL compiler is invoked via the command line with a synthesizing script passed as an argument to the compiler. This script sets up the 90 nm library, loads the VHDL files, synthesizes the design, and generates reports for the timing, cell usage, and power consumption. This script is also located in Appendix A.

The next major area is error analysis. This will be performed using MATLAB and Modelsim for simulations. The results from the MATLAB FFT function will be compared to those of the VHDL FFT implementation. Even though there is negligible error in the MATLAB due to the IEEE 754 floating point format, the MATLAB results will be the baseline for these tests. The twiddle factor bit width and it's effect on data will be analyzed. By varying the twiddle factor bit width, one can change the amount of error in the VHDL FFT compared to that of the MATLAB FFT function. The testing procedure for this is as follows: A *cosine* function is generated and the data points are sampled and placed into a text file. The FFT VHDL testbench opens this file and reads the data as inputs. The output is generated and placed into a different text file. MATLAB then resumes and reads in the VHDL FFT output data. A comparison of each output point is made between the MATLAB FFT function and the VHDL FFT function. A stem plot is generated showing the % error between both functions. In addition, statistics are generated for this set of data. This test is performed on all FFT lengths with varying twiddle factor bit widths. The MATLAB m-files which perform this testing are found in Appendix B. Additionally, a frequency sweep test will also be examined. A frequency ranging from 0 Hz to 2.5 GHz in steps of 10 MHz will be applied to the FFT. For each frequency step, a length-256 FFT will be calculated both in MATLAB and the VHDL FFT. The average and max percent error for a range of twiddle factor bit-widths and input bit-widths will be discussed.

3.9 Chapter Summary

The design of a configurable/dynamic FFT processor was discussed in this chapter. The overall design was introduced and then broken down into many smaller components. The design analysis was performed on each of these components, along with the different configurations of each. Each subcomponent was thoroughly tested in order to reduce the possible errors when assembling the final design. Additionally, testing procedures were developed to test both the error in the FFT calculations and the physical attributes such as power and timing.

IV. Analysis and Results

This chapter analyzes the results from the testing procedures discussed in the previous chapter. The data results will be examined first in the error analysis section. Next, the area and timing results from synthesis will be evaluated.

4.1 Error Analysis

4.1.1 Simple Cosine Curve. Due to the digital nature of the FFT algorithm and the use of approximate values for the twiddle factors, there will be error in the VHDL results as compared to those of the MATLAB FFT function. A comparison of twiddle factors with bit-widths of 6, 8, 10 ,and 12 are examined. A summary is shown in Figure 4.1 and figures 4.2 to 4.9 show detailed plots of the error between the VHDL and MATLAB FFT functions for each point of the N-length FFT. The error is calculated using the equation $error = \frac{VHDL-MATLAB}{MATLAB} * 100\%$. The analysis is performed on absolute values of the real and imaginary data outputs. The input bitwidth used is 10 bits. We will begin the analysis with the length-8 FFT as this is the first length to use twiddle factors in the calculations. An FFT of length 4 does not use twiddle factors, therefore there is no error between the VHDL and MATLAB data. The stem-and-leaf plot in Figure 4.2 shows the percent error between the MATLAB and VHDL values for each value of n. The results show relatively small error except for n values of 6 and 7. This is due to the twiddle factor approximation. For the n=6 case, the exact twiddle factor would be $(-cos(\frac{\pi}{4}),-j * sin(\frac{\pi}{4}))$. Expanding this out yields a results of $(-0.70710678, -0.70710678j)$. Because the twiddle factors in the VHDL implementation are scaled integer values, the result is scaled from a range of $(-1, 1)$ up to a range of $(-2^{tfbw-1}-1, 2^{tfbw-1}-1)$, where *tfbw* is the twiddle factor bit width. For a twiddle factor bit width of 10, this range becomes $(-511, 511)$. Scaling the n=6 twiddle factors results in values of $(-361.331565, -361.331565j)$. These values are ultimately rounded to $(-361, -361j)$. A negligible 2% error is introduced with this rounding. Increasing the twiddle factor bit width will reduce this error, but with an increase in chip area and power consumption. Figure 4.3 shows the plot of the FFT data and error analysis for a length-16 FFT. The results are similar to that of a length-8 FFT. With length-32 and length-64 FFTs, two sets of twiddle factors are now used. Because of this, the error in the first set of twiddle factors is now multiplied by using a second set of factors. This is evident in Figure 4.4 as the errors for each twiddle factor length are now larger than in the length-8 and length-16 case. This trend continues with Figures 4.5 - 4.9. The longer-length FFTs generate more error in the data than the shorter ones. The smaller width twiddle factor bit width used leads to a larger average error. Increasing the bit width from 6 to 8 shows a large decrease in average error. Increasing the bit width further to 10 yields better results, but beyond that the decrease in error is negligible.

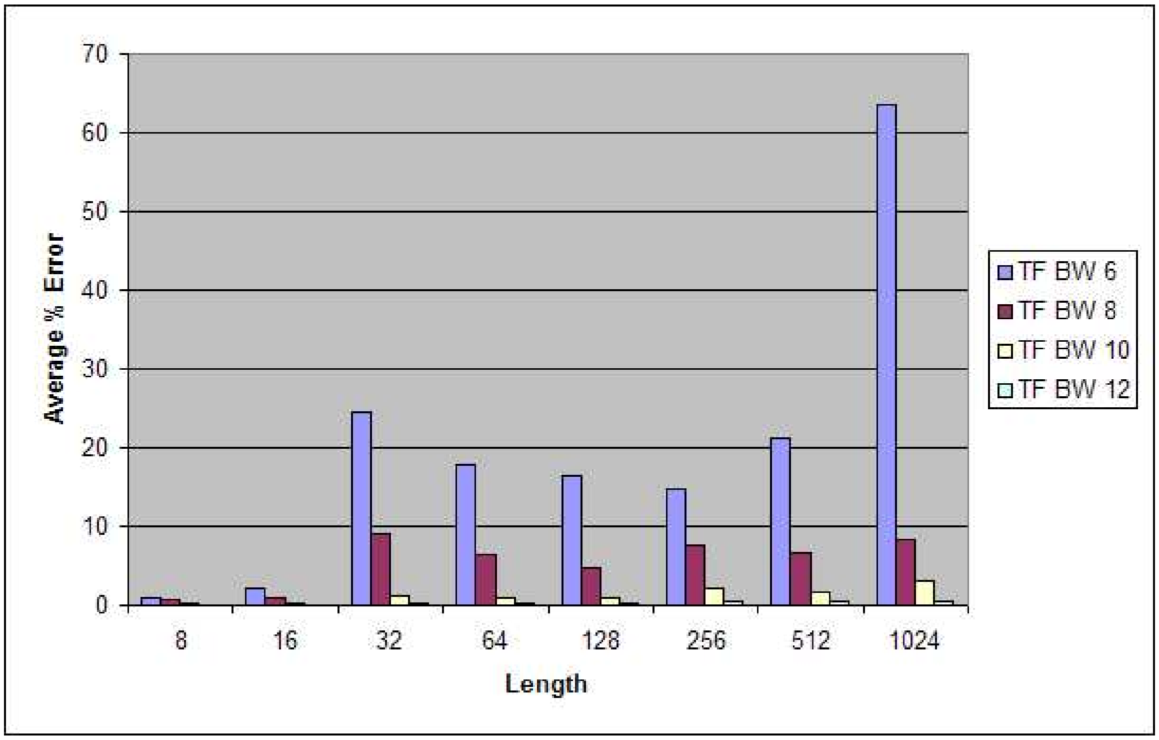


Figure 4.1: Average % Error

4.1.2 Frequency Sweep. For the frequency sweep, input bit-widths of 8, 10, 12, 14, and 16 along with twiddle factor bit-widths of the same values will be analyzed. The resulting frequency sweep produces a plot as shown in Figure 4.10. The average percent error and max percent error data is shown in Figures 4.11 and 4.12. The results for the maximum percent error show for a twiddle factor bit-width of 8 the

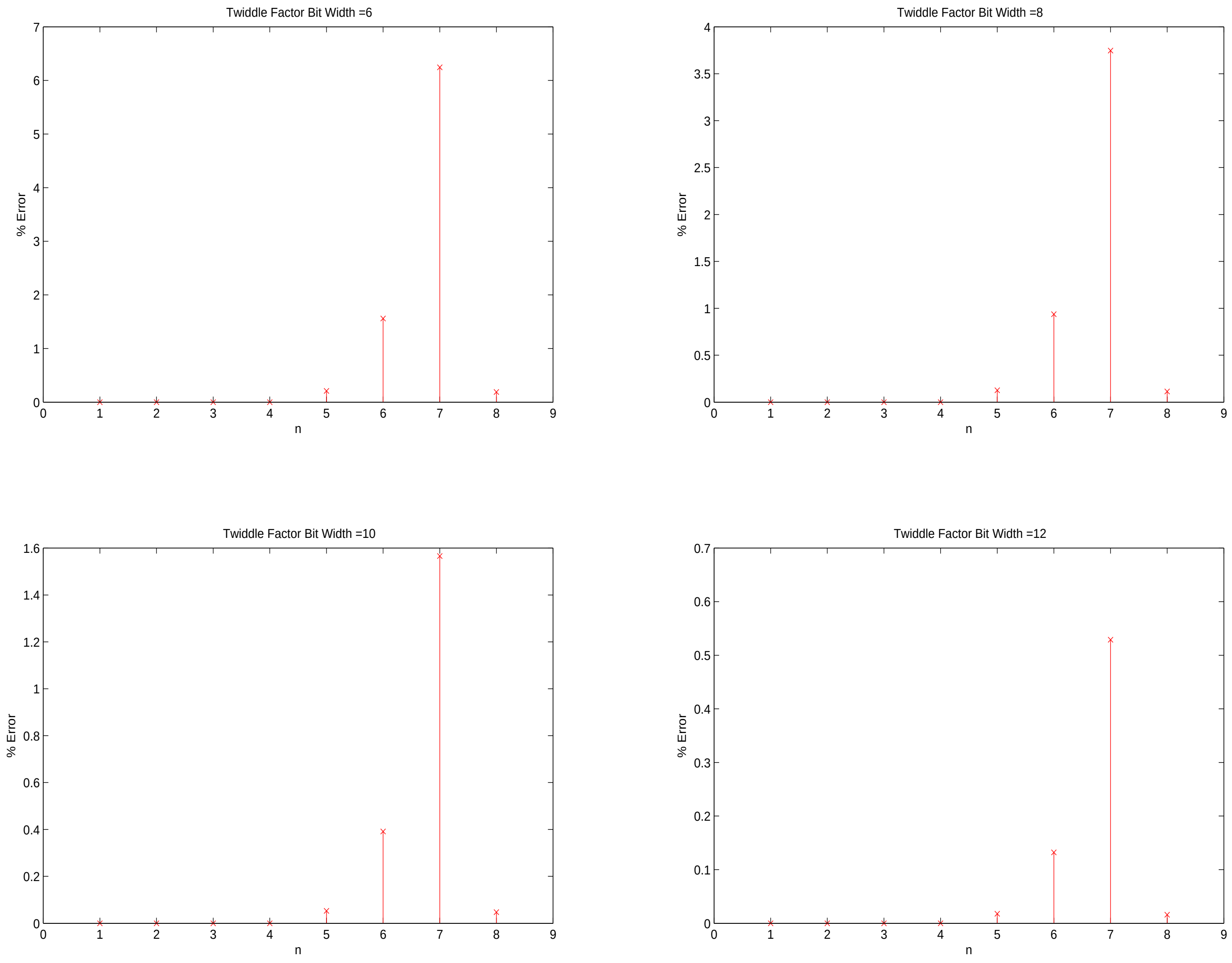


Figure 4.2: Percent error between MATLAB and VHDL FFT functions for N=8 and various twiddle factor bit-widths

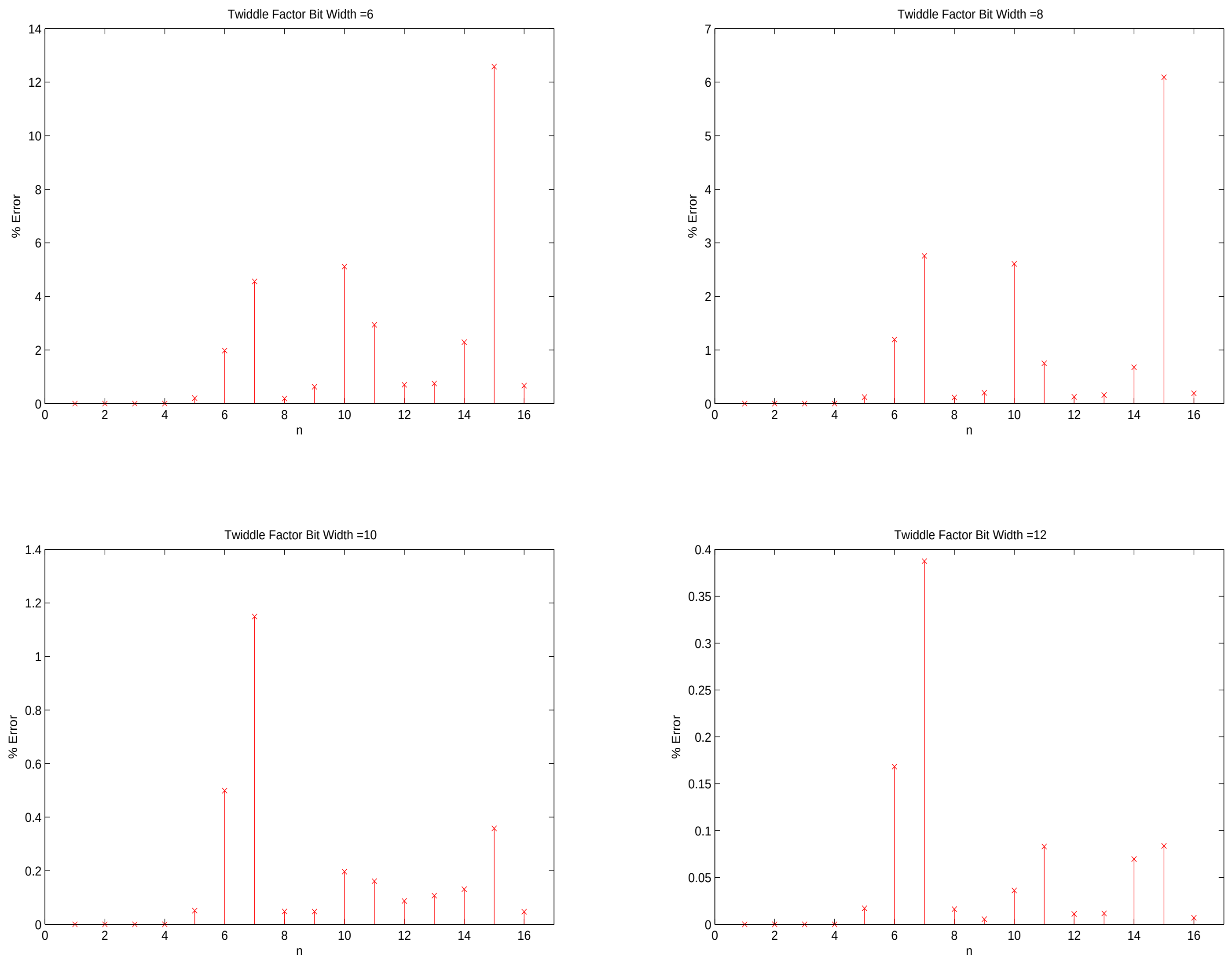


Figure 4.3: Percent error between MATLAB and VHDL FFT functions for N=16 and various twiddle factor bit-widths

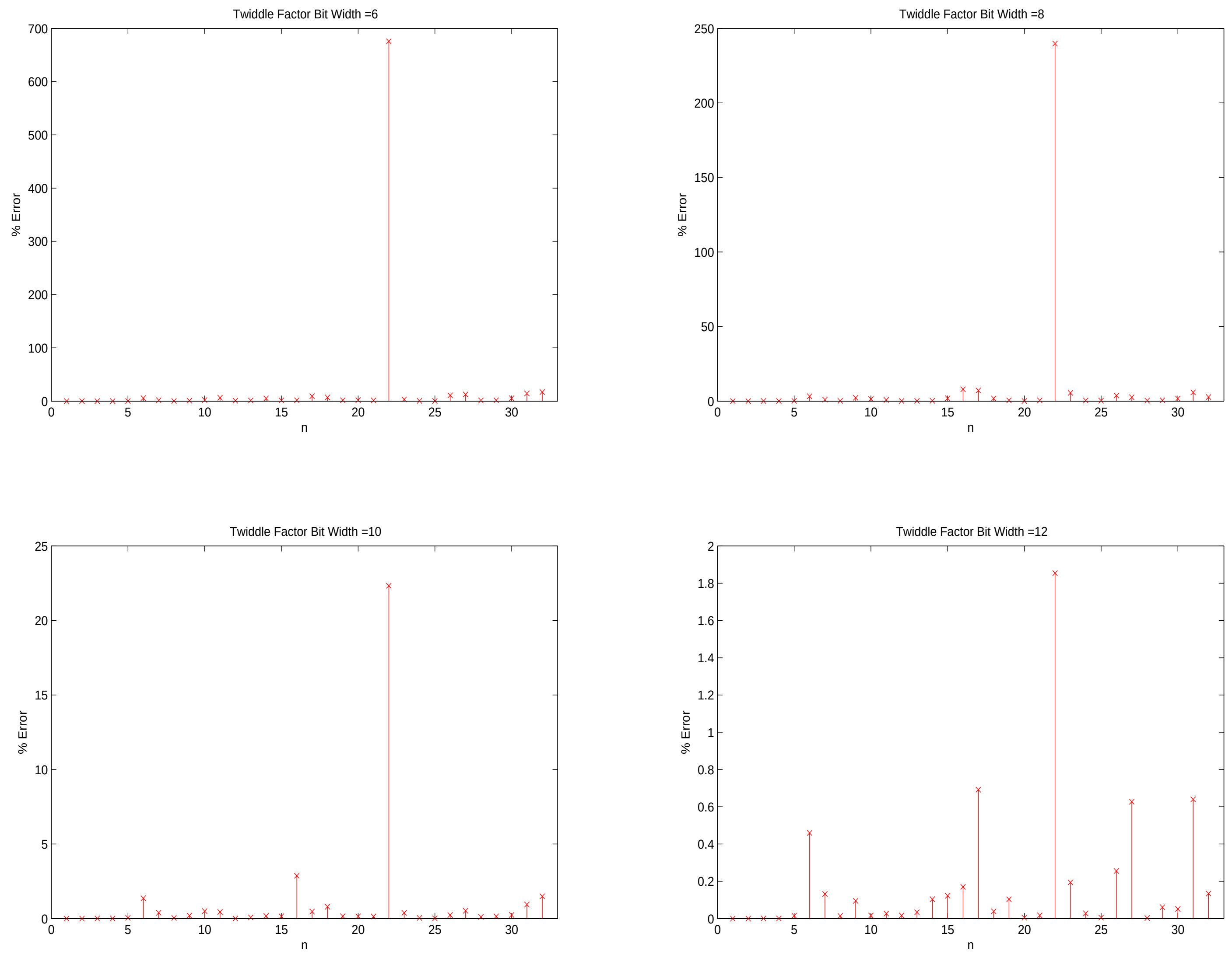


Figure 4.4: Percent error between MATLAB and VHDL FFT functions for N=32 and various twiddle factor bit-widths

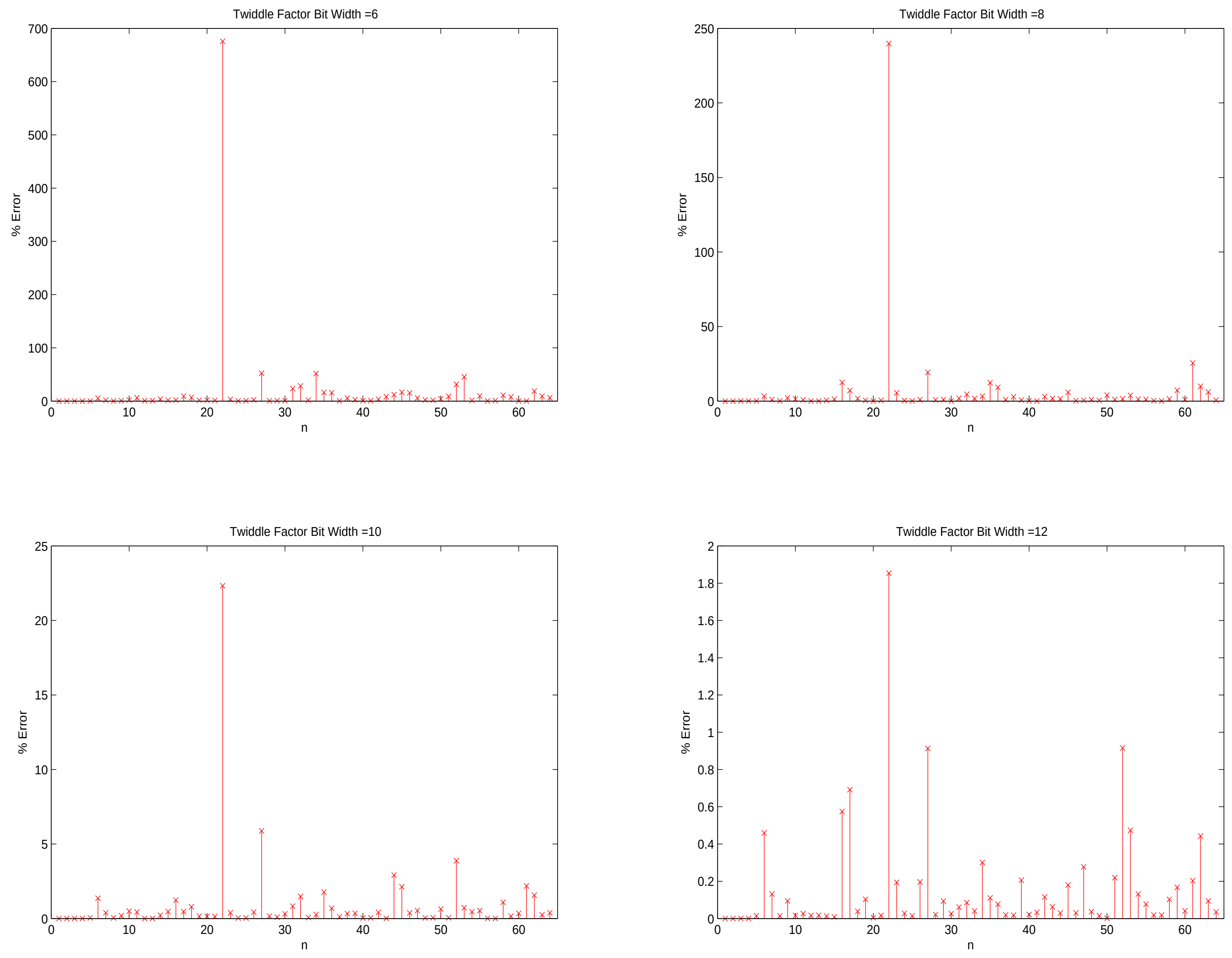


Figure 4.5: Percent error between MATLAB and VHDL FFT functions for N=64 and various twiddle factor bit-widths

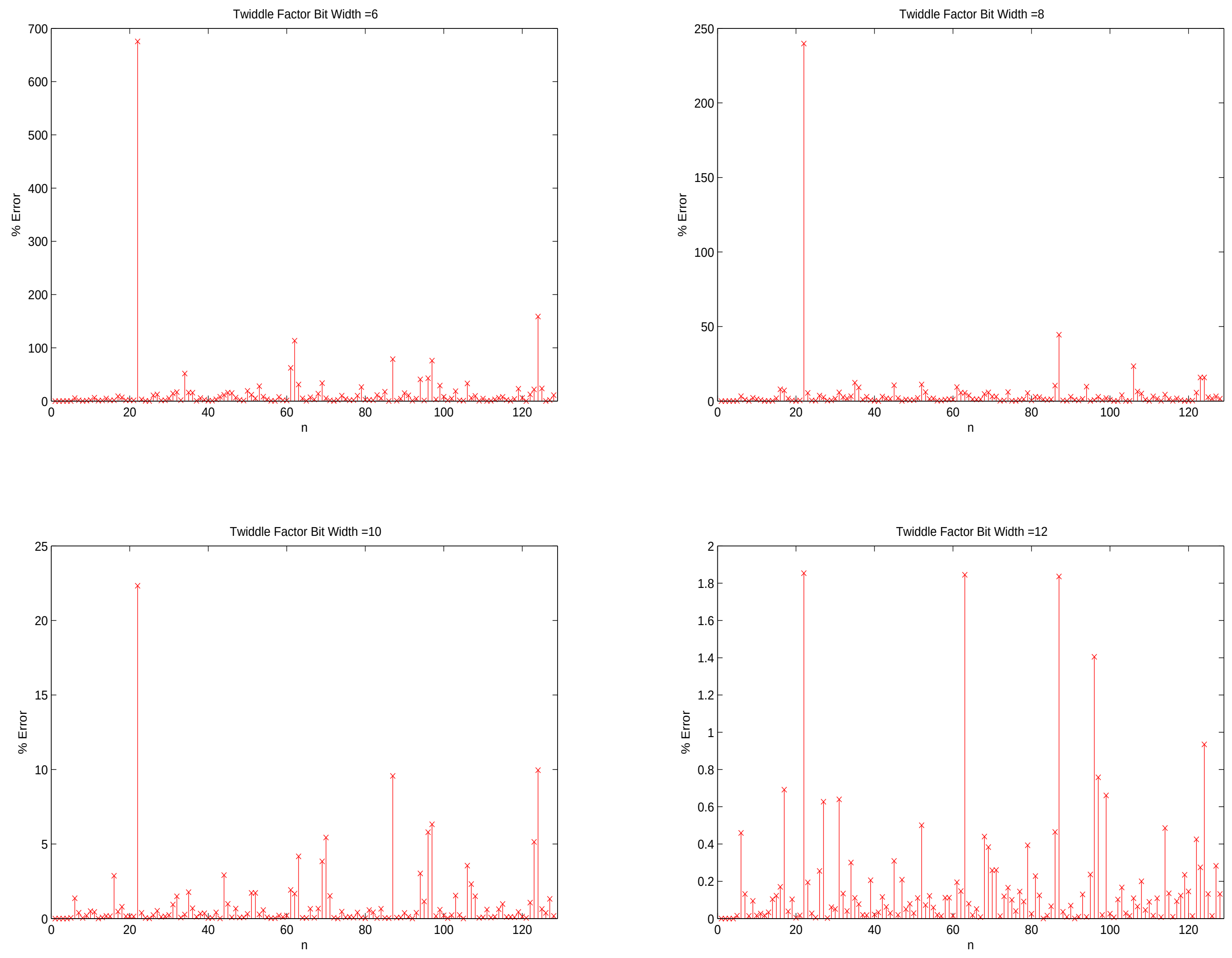


Figure 4.6: Percent error between MATLAB and VHDL FFT functions for N=128 and various twiddle factor bit-widths

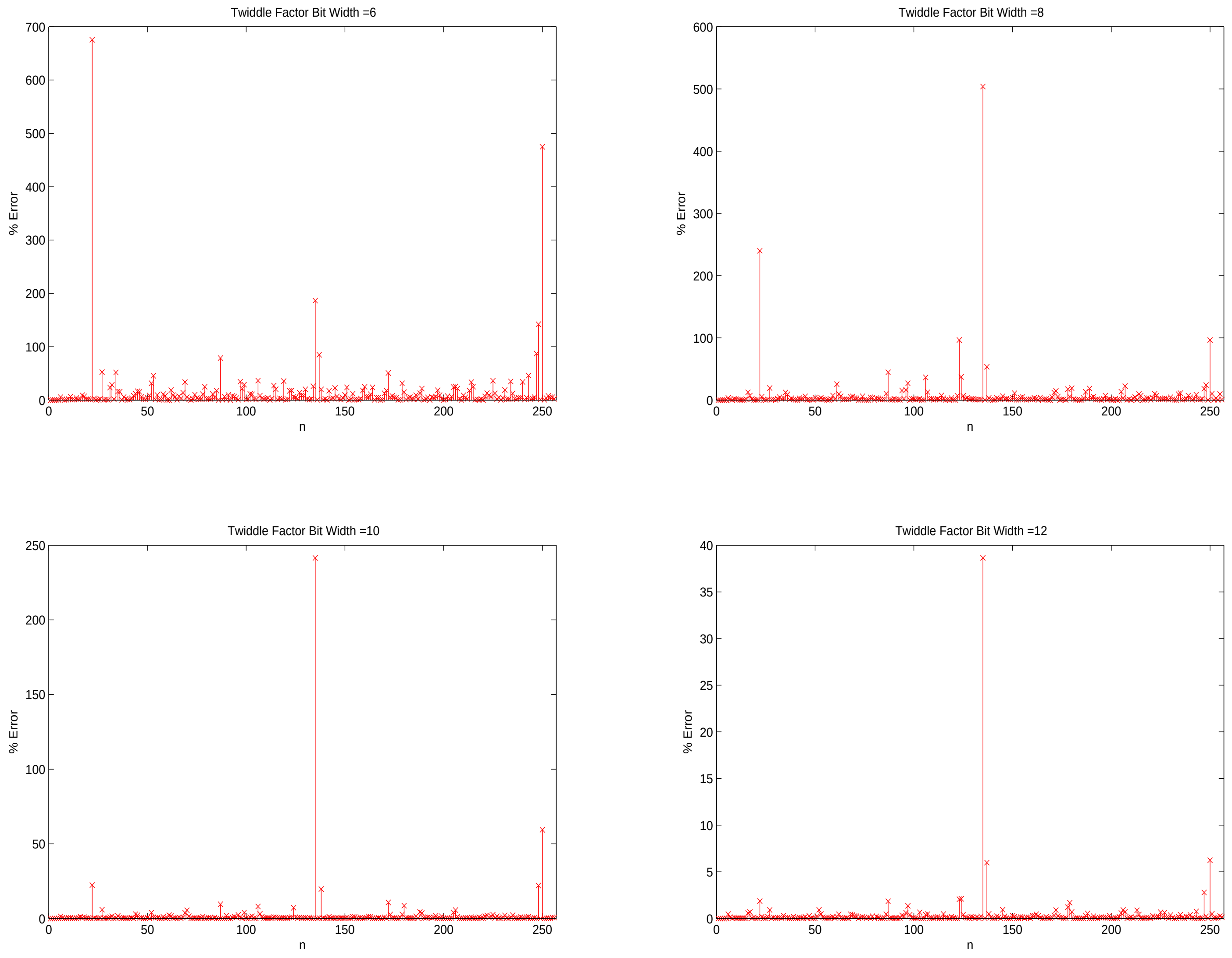


Figure 4.7: Percent error between MATLAB and VHDL FFT functions for N=256 and various twiddle factor bit-widths

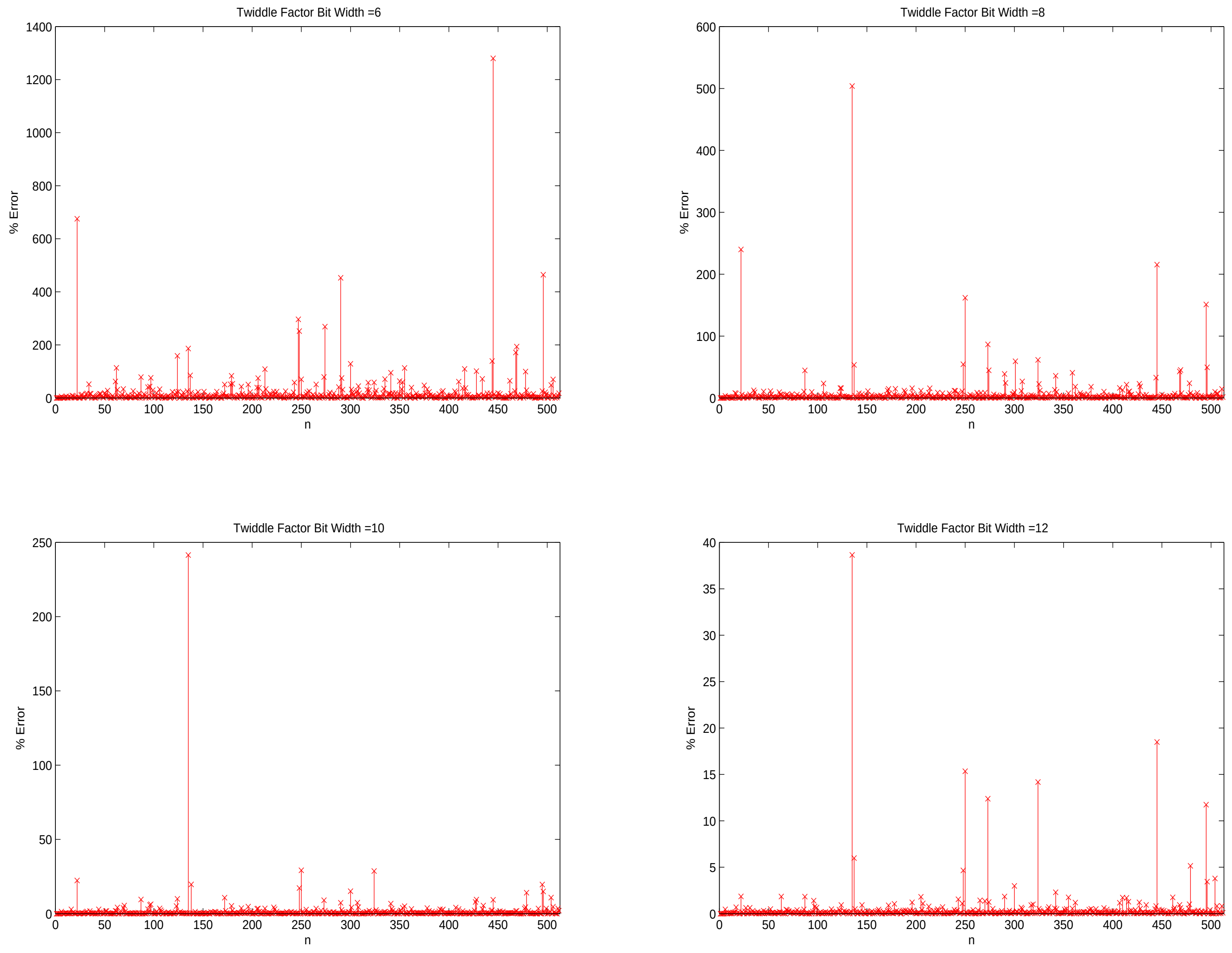


Figure 4.8: Percent error between MATLAB and VHDL FFT functions for N=512 and various twiddle factor bit-widths

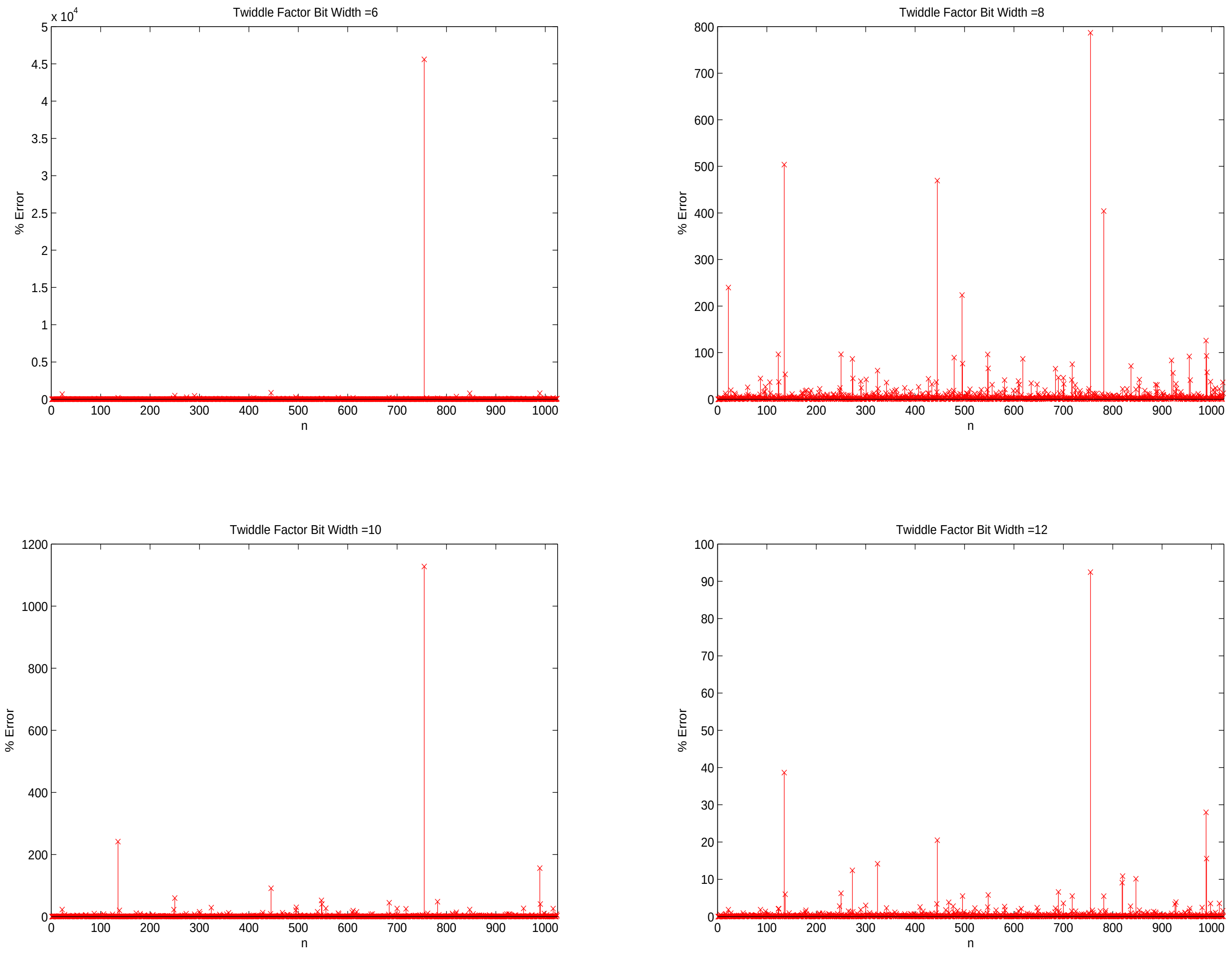


Figure 4.9: Percent error between MATLAB and VHDL FFT functions for N=1024 and various twiddle factor bit-widths

error remains the same no matter the input bit-width. This changes with twiddle factor bit-widths of 12 and higher. With an increase in input bit-width, maximum percent error drops for twiddle factor bit-widths of 10 - 16. The average percent error decreases with either an increase in twiddle factor bit-width and/or input bit-width.

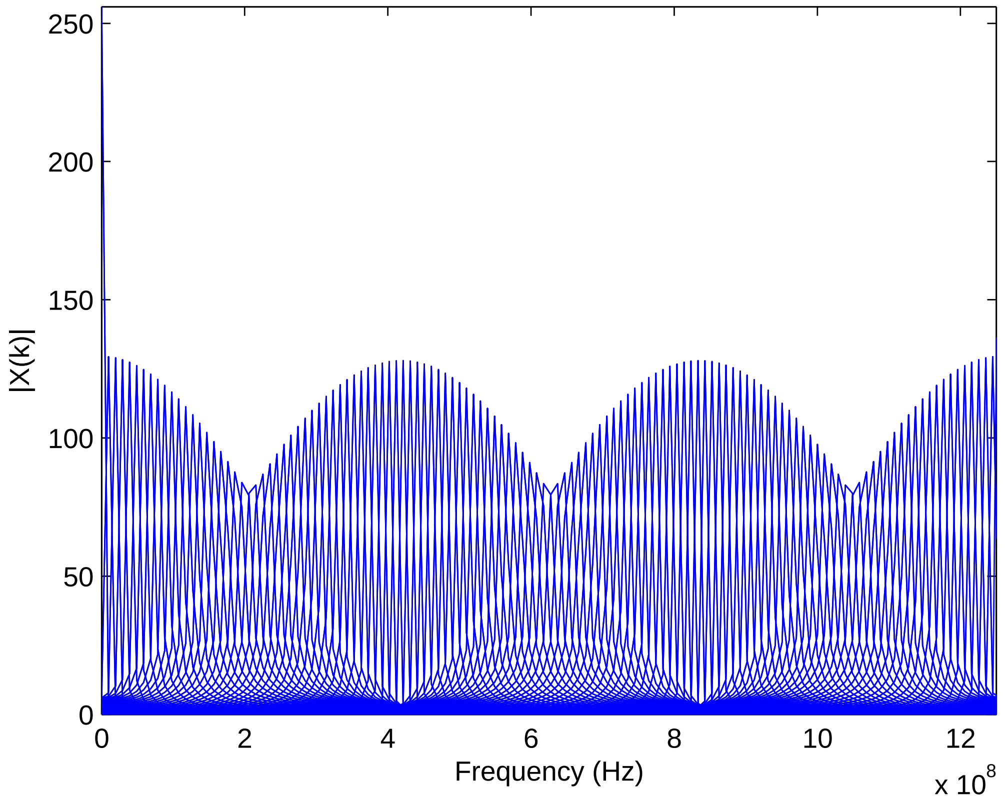


Figure 4.10: FFT Plot of Frequency Sweep for 0-2.5GHz

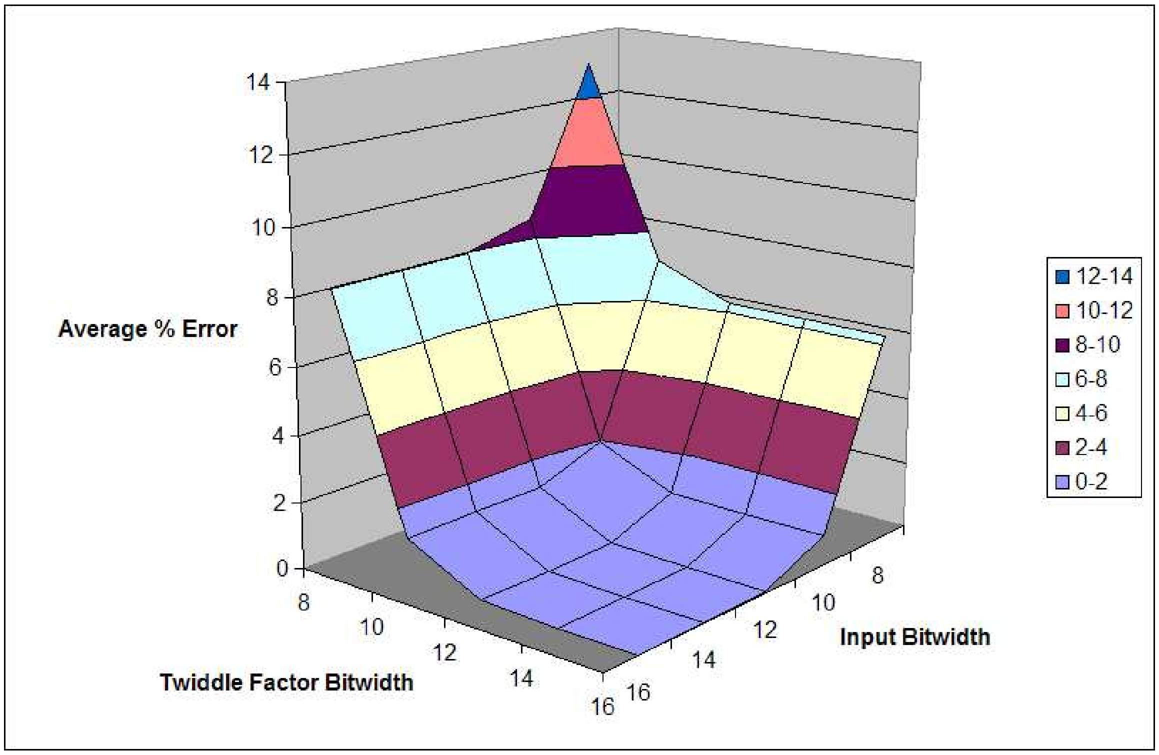


Figure 4.11: Average Error in Frequency Sweep

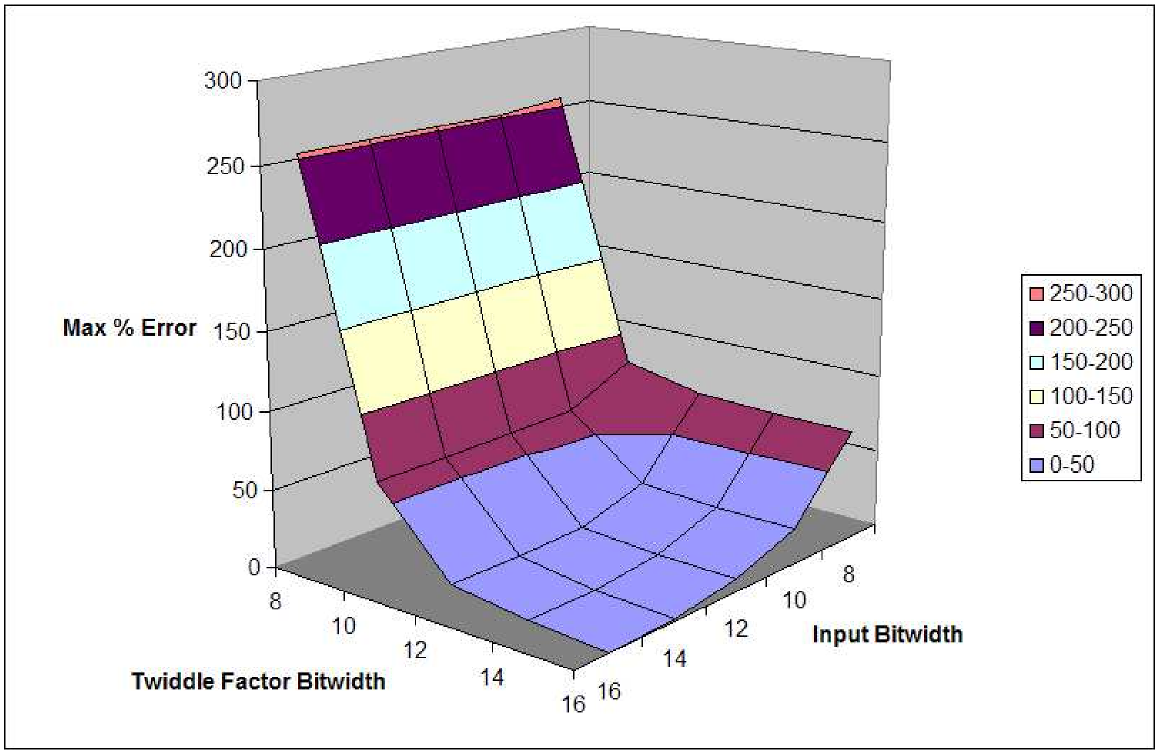


Figure 4.12: Maximum Error in Frequency Sweep

4.2 Timing Analysis

For this section and the following sections, an analysis will be performed on the physical effects of an FFT processor which can calculate one length (fixed) versus one which can calculate several lengths (dynamic). The critical path will be studied now. A comparison between the non-pipelined and pipelined versions is shown, using both a 350 nm and 90 nm technology. Using two different technology libraries will show the scaling for the timing analysis. Table 4.1 shows the parameters of the analysis. Figure 4.13 and 4.14 shows the results of the analysis. The y-axis shows the speed

Table 4.1: Parameters for Timing Analysis

Parameter	**Value**
log2N	3 - 10
nc	1 - 8 (based on log2N)
Input Width	10
Twiddle Factor Bit Width	10
Pipelining	Off/On
Software	Cadence Encounter RTL Compiler
Cell Libraries	TSMC High Performance General Purpose 90 nm AMI 350 nm

in MHz, while the x-axis is divided up by the different lengths (8 - 1024). Each of these is subdivided by the number of allowable dynamic sizes. A trend with the non-pipelined version shows maximum frequencies are similar for lengths using the same stages. For example, a length-128 and length-256 FFT use the same hardware, therefore the speeds are similar. Another trend is the larger the length, the smaller the maximum speed. This trend is due to large adders and multipliers which occur because the bit width of the data passing through the FFT is not rounded or clipped. Additionally, if a dynamic FFT is needed, any value of $nc > 1$ produces the same results. The pipelined version shows a doubling in maximum speed. For a length-8 FFT, speeds of approximately 450 MHz are obtained. On the other end of the spectrum, a length-1024 FFT can run between 200 - 250 MHz. The staggering values are due to various critical paths in the processor. The 90 nm version is approximately 6 times faster than the 350 nm.

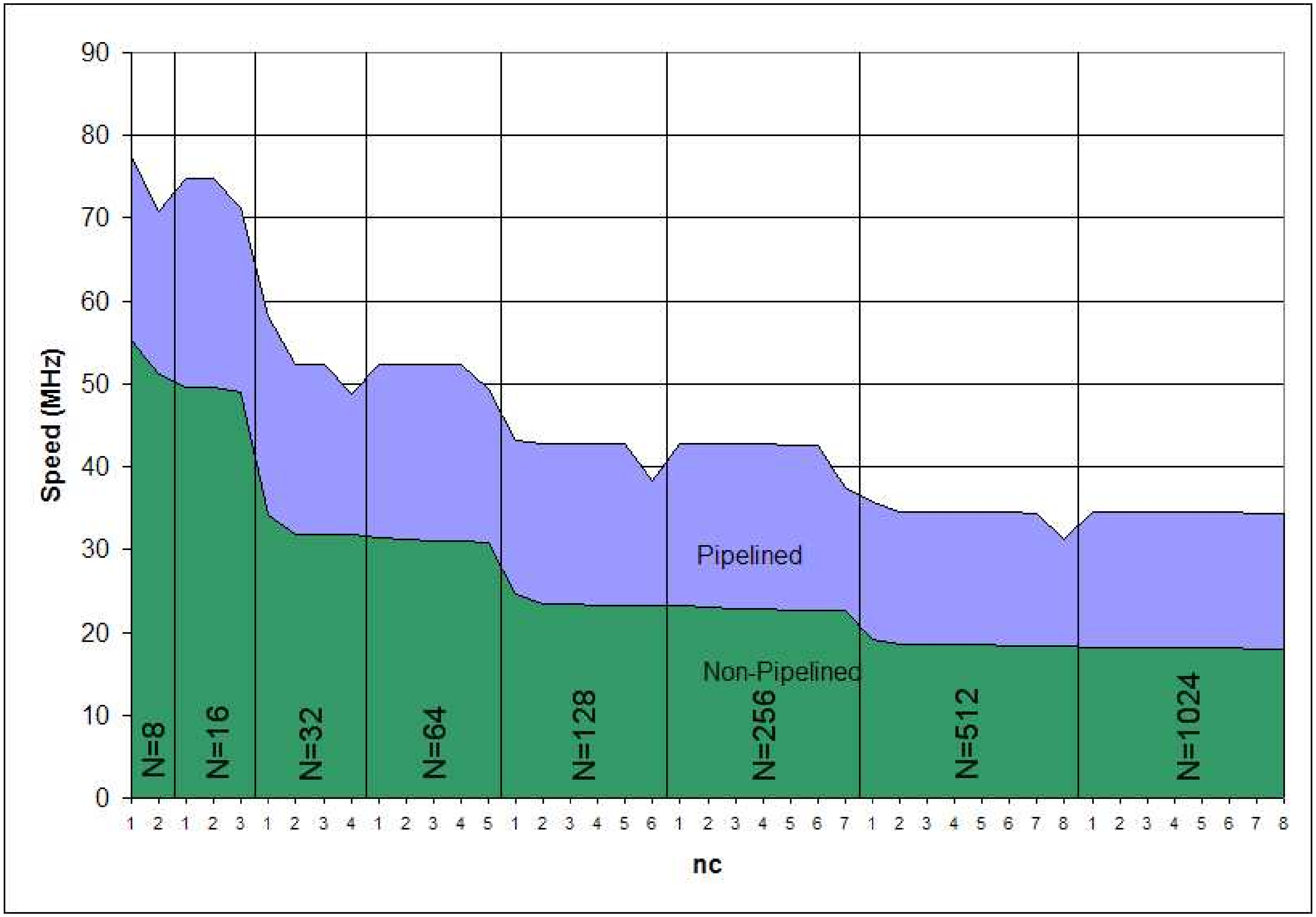


Figure 4.13: Maximum frequency for pipelined and non-pipelined FFTs with input bitwidth=10 and twiddle factor bitwidth=10 using the 350 nm technology.

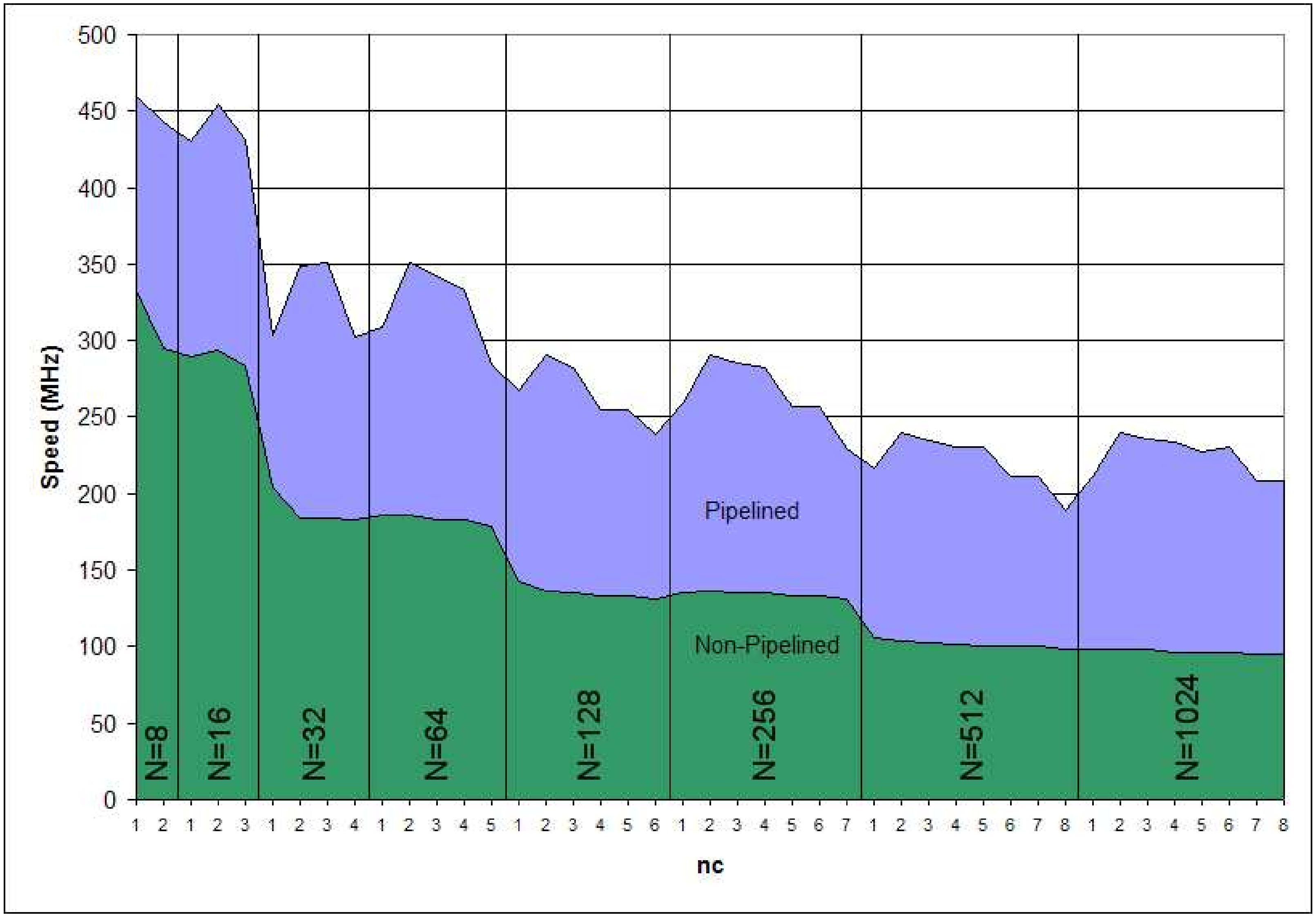


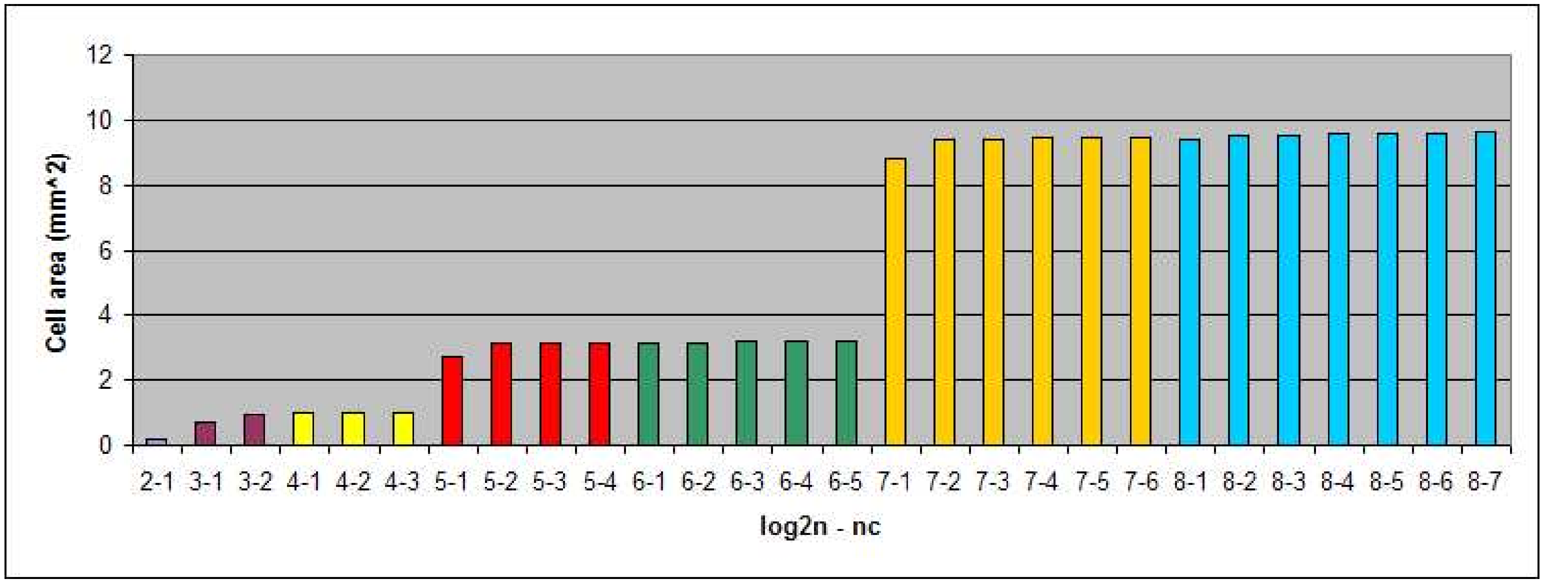
Figure 4.14: Maximum frequency for pipelined and non-pipelined FFTs with input bitwidth=10 and twiddle factor bitwidth=10 using the 90 nm technology.

4.3 Total Area Analysis

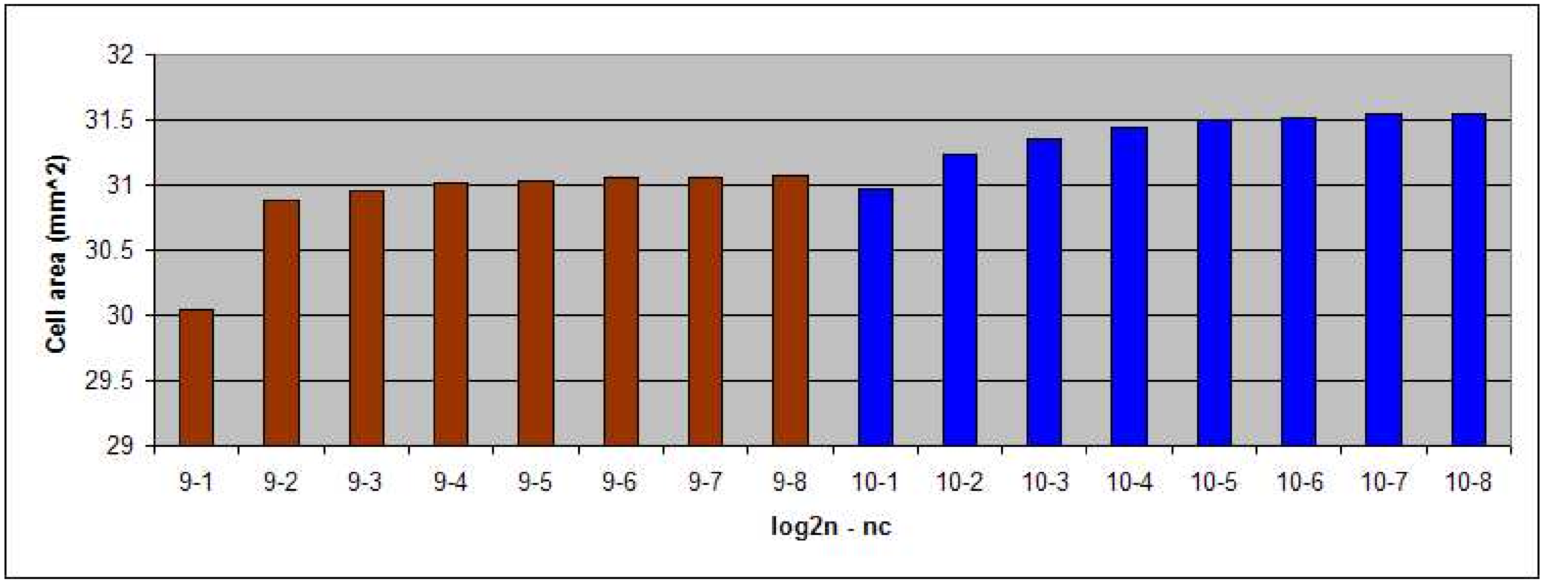
Table 4.2 shows the parameters for this testing. The total area needed for each configuration are now discussed, using both the 350 nm and 90 nm libraries. Figure 4.16 shows the results of the analysis. As expected, with each increase in length the total area increases. Total area for lengths using the same stages are similar, as in the case of the 32/64 lengths. The 90 nm version is approximately 44 times smaller in area than the 350 nm.

Table 4.2: Parameters for Total Area

Parameter	**Value**
log2N	2 - 10
nc	1 - 8 (based on log2N)
Input Bit Width	10
Twiddle Factor Bit Width	10
Pipelining	On
Software	Cadence Encounter RTL Compiler
Cell Libraries	TSMC High Performance General Purpose 90 nm AMI 350 nm

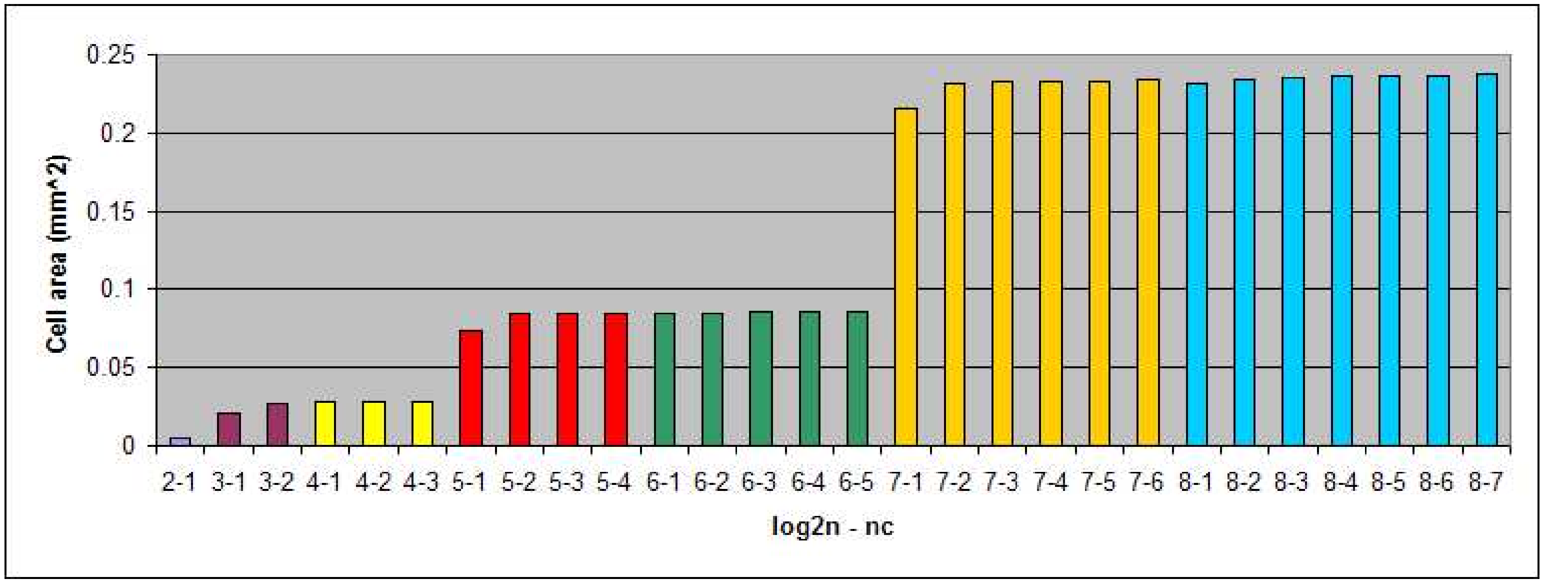


(a)

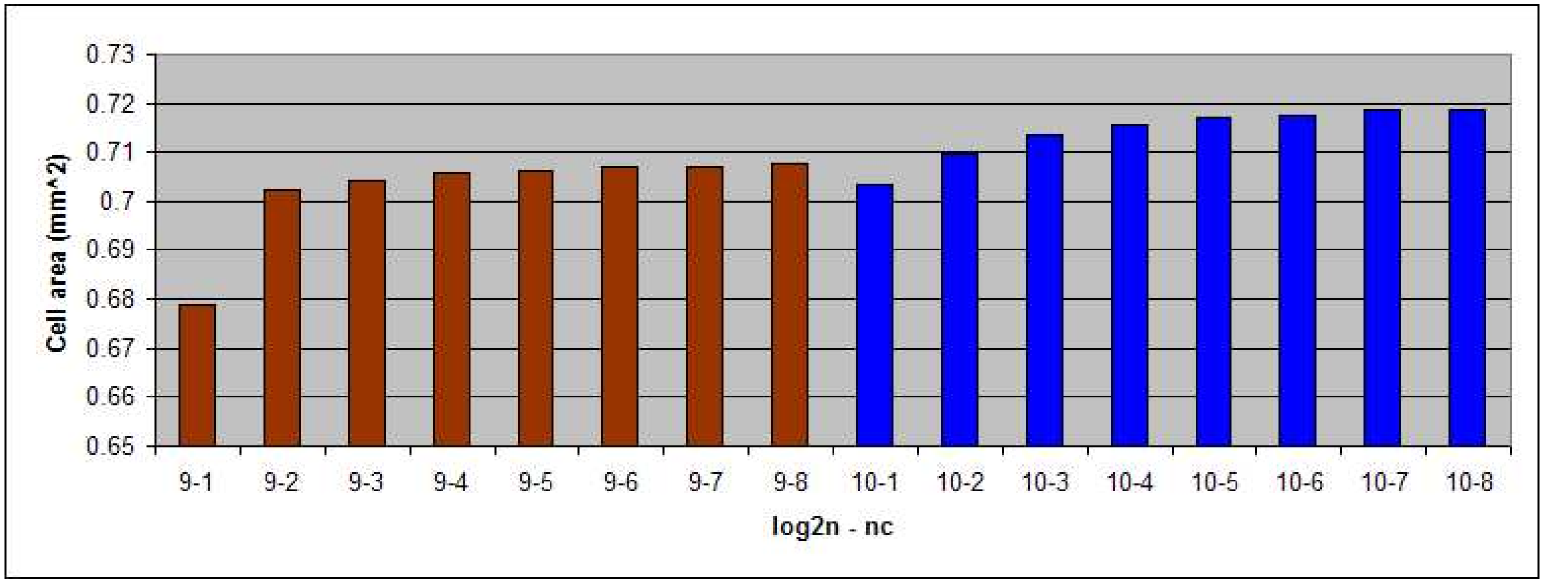


(b)

Figure 4.15: Total area using 350 nm technology for log2N from (a) 2 to 8 (b) 9 to 10



(a)



(b)

Figure 4.16: Total area using 90 nm technology for log2N from (a) 2 to 8 (b) 9 to 10

4.4 Chapter Summary

This chapter analyzed the results of the design. The error analysis shows the error resulting from different twiddle factor bit-widths compared to that of the FFT function found in MATLAB. Longer-length FFTs generally encounter more error than shorter lengths do. The frequency sweep shows how the input and twiddle factor bit-widths affect maximum and average percent error. Additionally, the timing and total area was analyzed for all possible configurations of the FFT processor. Increases in hardware for longer-length FFTs provide for an increase in total die area. Small changes are noticed with the increase in the dynamic size parameter *nc*.

V. Conclusions

5.1 Explanation of the Problem

The problem to be solved was to accomplish the characterization and implementation of a FFT using a fast and portable design strategy. By developing this type of strategy a designer can create digital components for specific functions in a matter of hours or days, as opposed to the conventional design flow which could take weeks or months. Developing standard components which are both configurable and dynamic and storing them in a library, will greatly decrease the development time for producing VLSI components for digital radar applications.

5.2 Summary of Background

A review of a variety of book, article, and internet sources was performed in order to understand, investigate, and verify the methods and previous technologies that support this research. A brief overview of the Fourier Transform, DFT, and FFT was discussed. Additionally, two algorithms in computing the FFT were examined. One of the algorithms, the DIF, was used in the implementation. Hardware descriptor languages, namely VHDL and Verilog, were discussed along with some pros/cons of each. Lastly, two previous implementations were analyzed.

5.3 HDL Code Development: Significance, Limitations, and Further Research

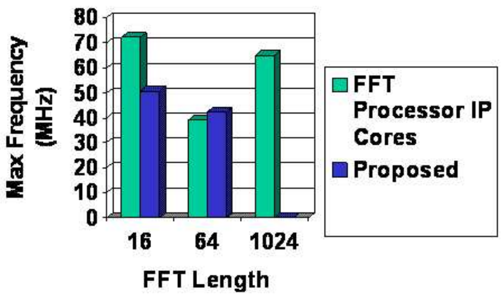
This research successfully demonstrated the use of a modular mixed signal VLSI design approach. An example component, the FFT, was developed and demonstrated for many types of configurations. The 90 nm technology library allowed a design to be synthesized using a smaller area and power consumption, in addition to faster speeds. In addition, the design was synthesized in a 350 nm library to show the scaling between the two technologies. The maximum speed at which this FFT processor can run is greatly enhanced also. In [11] the author demonstrates a speed of 20 MHz with a length-1024 FFT. In this research, speeds of approximately 225 MHz have been simulated, a speedup of nearly 1100%! Compared to the Handel-C version by [20] in which the author's implementation can attain a speed of 82 MHz, this paper's design is 274% faster. Figure 5.1 shows a comparison between this implementation and two other implementations [14] [15] on FPGAs. Although maximum frequency comparisons on similar hardware vary between +/- 10%, this implementation has its strength in being modular and portable. The VHDL is portable and self-contained, as the twiddle factors are generated from a source VHDL file. Additionally, because this function will be placed into a library, it is customizeable and dynamic. Before synthesizing, a designer can modify the FFT to be used in any type of project. Also, the dynamic properties of this FFT allow it to calculate different length FFTs during run-time with the simple modification of one signal.

Initially, designing configurable and dynamic components is a lengthy process. The total lines of VHDL code for the entire design is well over 11,000. All possible scenarios must be accounted for and tested. For this implementation, there are 44 possible configurations for max length and dynamic lengths. Adding pipelining options doubles this number to 88, which leads to a lengthy testing process. Once this is complete though, this design can be tailored to almost any specific need.

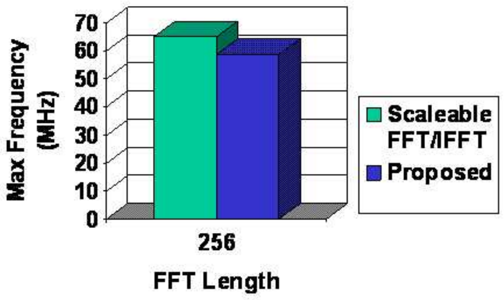
As with any type of computing device, there are several areas of research which can be expanded on to improve the FFT implementation. Several key areas to explore include expanding the 1024 length limit. With FFTs, the longer the length the more accurate the signal representations in the frequency domain. Also, a combination of the DIT and DIF algorithms will briefly be discussed as this will decrease the number of calculations needed.

5.3.1 Expanding Beyond the 1024-point Limit. Due to the modularity of the design, extending the maximum N value past 1024 is not difficult. A listing of the components which would need modification are listed below:

ROMGenerate.vhd The twiddle factors W_0 to W_3 were referenced with W_0 being the twiddle factor for the highest number stage (i.e. stage 5) in the design and W_3 being the factors for the second stage (the first stage does not use twiddle



(a)



(b)

Figure 5.1: Previous implementation comparisons

factors). This arranging of the variables is due to the way they are presented in the DIT and DIF algorithms. With this being said, to extend past 1024 the twiddle factor variables must be shifted. For example, if one wanted to use a max length of 4096, the twiddle factors would range from W_0 to W_4 with the new stage 6 using W_0. The only changes needed for the *ROMGenerate.vhd* code would be to the main function. Here, one would change the *Mp_array* variable, the p for loop, and the *Mp* calculation.

FFT.vhd To add 2048 as a possible length, a new generate statement will be needed (i.e. *Neq2048*). By following the previous size implementation, it is easy to see the pattern. All the possible *nc* choices will have to be covered also. This allows the minimal components necessary for each value of *nc*. It is helpful to create a drawing similar to the completed design layout shown in Chapter II to implement the necessary muxes and sign extenders. Karnaugh maps are very useful here.

Components.vhd To follow along with the design by He and Torkelson [10], the twiddle factors are numbered from 0 to 3 going from left to right in the design. To accommodate a larger length, these values must be 'shifted' to the left. Renumbering the twiddle factors in this module and adding a new *W4* component will work.

All other components are modular and do not need any modifications.

5.3.2 Implementation of a Decimation-In-Time-Frequency Algorithm. Ali Saidi developed an algorithm which he claims reduces the number of real multiplications and additions [17]. The algorithm is called "Decimation-In-Time-Frequency (DITF) FFT Algorithm." This reduces the arithmetic complexity while using the same structure as the conventional Cooley-Tukey FFT algorithm. He extended the algorithm to the radix-2 FFT implemented in this research. The author explains the heart of the DITF algorithm is based on this observation: in the DIF algorithm most of the calculations are performed in the early stages of the algorithm while in the DIT algorithm most of the calculations are done in the final stages of the algorithm [17]. The author proposes starting with the DIT FFT algorithm and then switching to the DIF FFT algorithm as some intermediate stage will decrease the amount of computations needed. The flow graph in Figure 5.2 illustrates a 32-point DITF FFT algorithm. The cost of the transition from DIT to DIF and the savings due to this transition vary depending on the stage at which the algorithms switch. An analysis is performed by the author in this article [17].

Table 5.1 shows the number of real multiplies for several lengths (N) for both the Radix-2 Cooley-Tukey and the DITF algorithm, along with several other algorithms. The data verifies the number of multiplications is smaller for the DITF algorithm, especially for larger lengths. By decreasing the number of operations necessary to compute the FFT, the calculation overall will be performed faster.

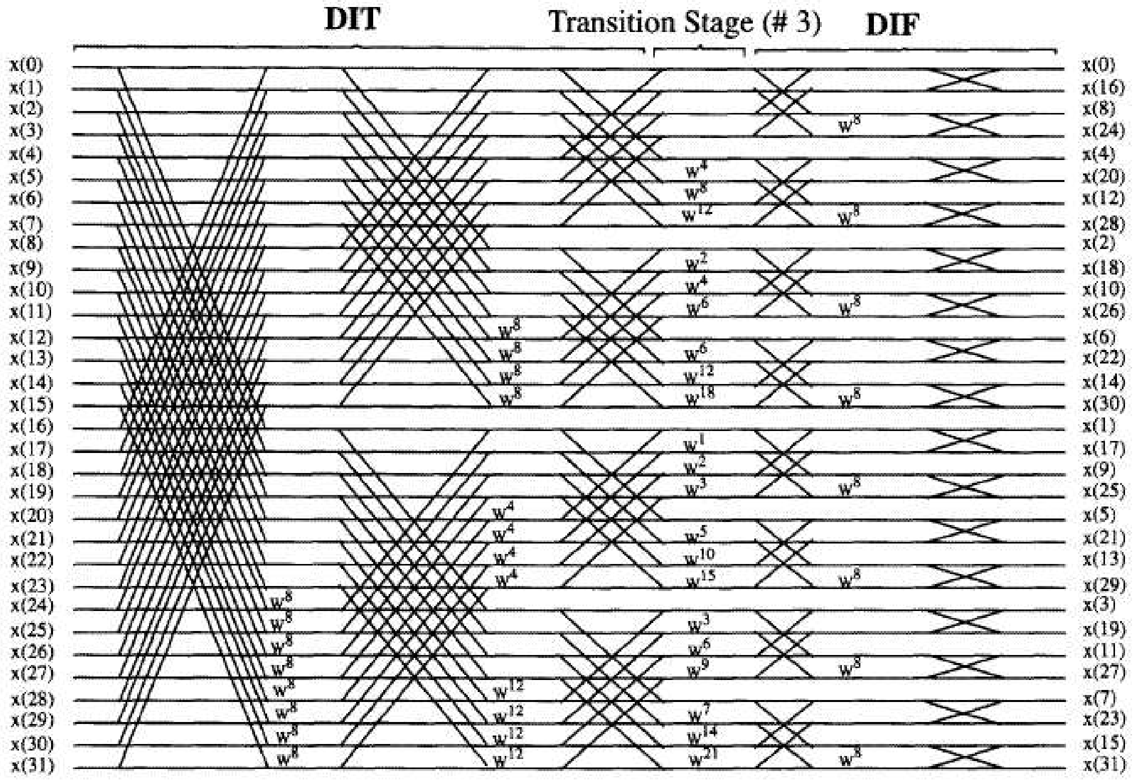


Figure 5.2: 32-Point DITF FFT Flow Graph [17]

Table 5.1: Number of real multiplies for complex FFT algorithms. [17]

Size		Split	Radix-2	Radix-2	Radix-4	Radix-4	Radix-8	Radix-8
M	N	RADIX	CT	DITF	CT	DITF	CT	DITF
3	8	4	4	4	N/A	N/A	N/A	N/A
4	16	24	28	24	24	24	N/A	N/A
5	32	84	108	88	N/A	N/A	N/A	N/A
6	64	248	332	248	264	264	248	248
7	128	660	908	696	N/A	N/A	N/A	N/A
8	256	1656	2316	1784	1800	1656	N/A	N/A
9	512	3988	5644	4472	N/A	N/A	3992	3992
10	1024	9336	13324	10744	10248	9528	N/A	N/A
11	2048	21396	30732	25336	N/A	N/A	N/A	N/A
12	4096	48248	69644	58360	53256	49656	48280	47608

Appendix A. FFT Synthesis Testings Scripts

Listing A.1: PERL Synthesis Script

```perl
#!/usr/bin/perl -w

# Make sure the files we need exist...
if(!-e "FFT.vhd") {
        print "Missing FFT.vhd\n";
        exit();
}
if(!-e "FFTComponents.vhd") {
        print "Missing FFTComponents.vhd\n";
        exit();
}
if(!-e "ROMGenerate.class") {
        print "Missing ROMGenerate.class\n";
        exit();
}
if(!-e "FFT.cmd") {
        print "Missing FFT.cmd\n";
        exit();
}

# For this set of tests, fix the input bitwidth and
# the twiddle factor bitwidth to 10 bits each.
$in_width = 10;
$tf_width = 10;

# Loop through all possible lengths...
for($log2N = 2 ; $log2N <= 10 ; $log2N++) {
        # determine the max nc value.
        $maxNC = $log2N - 1;
        if($log2N == 10) {
                $maxNC = 8;
        }

        # loop through all possible nc values
        for($nc = 1 ; $nc <= $maxNC ; $nc++) {
                print "Synthesizing $log2N $nc $tf_width $in_width\n";

                # calculate the output width, which is based on
                # log2N, input_width, and tf_width
                $temp = $log2N + ($log2N % 2);
                $out_width = $in_width + $temp * 1 + ($temp/2 - 1) * $tf_width;

                # Copy FFT_template.vhd to FFT.vhd
                `cp FFT_template.vhd FFT.vhd`;

                # Modify the parameters of the FFT.vhd file...
                print "  Modifying FFT.vhd...\n";
                `perl -pi -e 's/#in_width/$in_width/g' FFT.vhd`;
                `perl -pi -e 's/#out_width/$out_width/g' FFT.vhd`;
                `perl -pi -e 's/#tf_width/$tf_width/g' FFT.vhd`;
                `perl -pi -e 's/#nc/$nc/g' FFT.vhd`;
                `perl -pi -e 's/#logbase2N/$log2N/g' FFT.vhd`;
                `perl -pi -e 's/#pipeline/YES/g' FFT.vhd`;

                # Generate the twiddle factors
                print "  Generating TwiddleFactors.vhd...\n";
                `~/bin/java ROMGenerate $log2N $nc $tf_width vhdl`;
        # If log2N = 2 (length 4) create a blank TwiddleFactors.vhdl
        # so Cadence won't choke
        if($log2N == 2)  {
                `touch TwiddleFactors.vhd`;
        }

        # execute RTL script...
        print "  Running synthesis script...\n";
        `rc -files FFT.cmd`;

        # copy output files to specific file
        print "  Copying results to synth_results directory...\n";
        $timing_filename=join '','timing_',$log2N,'_',$nc,'_PIPE.txt';
        $area_filename=join '','area_',$log2N,'_',$nc,'_PIPE.txt';
        $power_filename=join '','power_',$log2N,'_',$nc,'_PIPE.txt';
        `mv timing.txt synth_results/$timing_filename`;
        `mv area.txt synth_results/$area_filename`;
        `mv power.txt synth_results/$power_filename`;
    }
}
```

Listing A.2: Cadence Synthesis Script

```
# Cadence RTL Compiler (RC)
#   version 05.20-p002 (32-bit) built Nov 28 2005
#
# Run with the following arguments:
#   -logfile rc.log
#   -cmdfile rc.cmd

# setup the library search path to the 90nm libraries from TSMC
set_attribute lib_search_path /home/afiten3/gce07m/bbrakus/libraries/TSMCHOME/digital/Front_End/...
    timing_power/tcbn90ghp_150a

# setup the hdl search paths to the current directory
set_attribute hdl_search_path .

# load one of the 90nm libraries
set_attribute library  tcbn90ghpbc.lib

# read all the vhdl files
read_hdl -vhdl TwiddleFactors.vhd FFTComponents.vhd FFT.vhd

# compile and check for errors
elaborate FFT

# synthesize the design
synthesize -to_mapped FFT

# create reports and save them to the current directory
report timing > timing.txt
report area > area.txt
report power > power.txt

quit
```

Listing B.1: MATLAB Error Analysis Script

```
function [  ] = TestFFTerror(log2length, input_bitwidth)
%TestFFTerror Generates test input and FFT results data
%  TestFFTerror(log2length, input_bitwidth)
%       log2length = log base 2 of FFT length
%       input_bitwidth = bitwidth of input data

length = 2^log2length;
in_scale = 2^(input_bitwidth-1)-1;
n=[0:29];
data=cos(2*pi*n/10);
data=round(data * in_scale);

% open file to store input data
in_id=fopen('input_data.txt','wt');
if(in_id == -1)
    error('cannot open file for writing');
end
% store input_data in file...
for j=1:30
    fprintf(in_id, '%d\n', real(data(j)));
    fprintf(in_id, '%d\n', imag(data(j)));
end
for j=31:length
    fprintf(in_id, '0\n');
    fprintf(in_id, '0\n');
end
fclose(in_id);

for tfbw=6:2:12
    % scale factor
    tf_scale = 2^(tfbw-1)-1;

    % calculate FFT of data...
    matlab=(fft(data,length));

    % scale data based on length...
    if(length == 8 || length == 16)
        matlab = matlab * tf_scale^1;
    elseif(length == 32 || length == 64)
        matlab = matlab * tf_scale^2;
    elseif(length == 128 || length == 256)
        matlab = matlab * tf_scale^3;
    elseif(length == 512 || length == 1024)
        matlab = matlab * tf_scale^4;
    end

    %bit reverse data...
    rev=zeros(1,length);
    for j=0:length-1
        binstr = dec2bin(j, log2(length));
        binstr = fliplr(binstr);
        bitrev = bin2dec(binstr);
        rev(bitrev+1) = matlab(j+1);
    end

    % open file to store matlab FFT data
    fftfilename = strcat('fft_matlab_', num2str(length), '_', num2str(tfbw), '.txt');
    fft_id=fopen(fftfilename, 'wt');
    if(fft_id == -1)
        error('cannot open file for writing');
    end
    % store FFT data in file...
    for j=1:length
        fprintf(fft_id, '%d %d\n', round(real(rev(j))), round(imag(rev(j))));
    end
    fclose(fft_id);

    disp('Run the Modelsim simulator to generate VHDL data using the following parameters:');
    dispstr=strcat('input_width = ' , num2str(input_bitwidth));
    disp(dispstr);
    temp = log2length + mod(log2length,2);
    output_width = input_bitwidth + temp*1 + (temp/2 -1)*tfbw;
    dispstr=strcat('output_width = ' , num2str(output_width));
    disp(dispstr);
    dispstr=strcat('tfbw = ', num2str(tfbw));
    disp(dispstr);
    dispstr=strcat('log2N = ',num2str(log2length));
    disp(dispstr);
    disp(' ');
    disp('Press any key when done...');
    pause;
    % rename the generated VHDL FFT data
    vhdlfilename = strcat('fft_vhdl_',num2str(length), '_', num2str(tfbw), '.txt');
    movefile('fft_vhdl.txt',vhdlfilename);
end
CompareDataStem(log2length);
```

Listing B.2: MATLAB Stem Plot Script

```
function [  ] = CompareDataStem(log2N)
%CompareDataStem Compares error results and produces a stem plot
%  Detailed explanation goes here
n=2^log2N;

for tfbw=6:2:12
    matlabFilename=strcat('fft_matlab_', int2str(n), '_', int2str(tfbw), '.txt');
    vhdlFilename=strcat('fft_vhdl_', int2str(n), '_', int2str(tfbw), '.txt');
    matlabID = fopen(matlabFilename, 'r');
    diffFilename=strcat('fft_diff_',int2str(n), '_',int2str(tfbw), '.txt');
    vhdlID = fopen(vhdlFilename, 'r');
    diffID = fopen(diffFilename, 'wt');

    if (diffID == -1)
        error('cannot open file for writing');
    end

    matlabTHM = fscanf(matlabID, '%f %f', [2 inf]);
    vhdlTHM = fscanf(vhdlID, '%f %f', [2 inf]);
    matlabTHM=matlabTHM';
    vhdlTHM=vhdlTHM';

    matlabRE = (matlabTHM(:,1));
    matlabIM = (matlabTHM(:,2));
    vhdlRE = (vhdlTHM(:,1));
    vhdlIM = (vhdlTHM(:,2));

    n=length(matlabRE);
    reDiff=zeros(1,n);
    imDiff=zeros(1,n);
    for i=1:n
        if(matlabRE(i) ~= 0 && matlabIM(i) ~= 0)
            reDiff(i) = (100*(vhdlRE(i)-matlabRE(i))/matlabRE(i));
            imDiff(i) = (100*(vhdlIM(i)-matlabIM(i))/matlabIM(i));
        else
            reDiff(i) = 0;
            imDiff(i) = 0;
        end

        fprintf(diffID, '%f %f\n', reDiff(i), imDiff(i));
    end
    reMaxError = max(reDiff);
    reMinError = min(reDiff);
    imMaxError = max(imDiff);
    imMinError = min(imDiff);

    titlestring=strcat('Twiddle Factor Bit Width =  ',int2str(tfbw));
    subplot(2,2,(tfbw-4)/2);
    if((tfbw-4)/2 == 1)
        stem([1:n],reDiff, 'bx'),xlim([0 n+1]);
        hold on
        stem([1:n],imDiff, 'r+'),title(titlestring),xlabel('n'),ylabel('% Error'),xlim([0 n+1]);
    elseif((tfbw-4)/2 == 2)
        stem([1:n],reDiff, 'bx');,xlim([0 n+1]);
        hold on
        stem([1:n],imDiff, 'r+'),title(titlestring),xlabel('n'),ylabel('% Error'),xlim([0 n+1]);
    elseif((tfbw-4)/2 == 3)
        stem([1:n],reDiff, 'bx');,xlim([0 n+1]);
        hold on
        stem([1:n],imDiff, 'r+'),title(titlestring),xlabel('n'),ylabel('% Error');,xlim([0 n+1]);
    else
        stem([1:n],reDiff, 'bx');,xlim([0 n+1]);
        hold on
        stem([1:n],imDiff, 'r+'),title(titlestring),xlabel('n'),ylabel('% Error'),xlim([0 n+1]);
    end

    fclose(matlabID);
    fclose(vhdlID);
    fclose(diffID);

    % open file to store error data
     errorfilename = strcat('images\N', num2str(n), 'TF', num2str(tfbw), 'error.txt');
     error_id=fopen(errorfilename, 'wt');
     if(error_id == -1)
         error('cannot open file for writing');
     end
     reMaxError = max(abs(reDiff));
     reAveError = average(reDiff);
     reStdDev = std(reDiff);
     imMaxError = max(abs(imDiff));
     imAveError = average(imDiff);
     imStdDev = std(imDiff);
     fprintf(error_id, 'Max real error = %f\n', reMaxError);
     fprintf(error_id, 'Average real error = %f\n', reAveError);
     fprintf(error_id, 'Standard deviation of real = %f\n', reStdDev);
     fprintf(error_id, 'Max imag error = %f\n', imMaxError);
     fprintf(error_id, 'Average imag error = %f\n', imAveError);
     fprintf(error_id, 'Standard deviation of imag = %f\n', imStdDev);
     fclose(error_id);
end
```

Bibliography

1. "Celoxica". URL `www.celoxica.com`.
2. Arfken, G. *Mathematical Methods for Physicists.* Academic Press, Orlando, FL, 3rd edition, 1985.
3. Boeing. *Boeing MSP Macro Databook.*
4. Bracewell, R. *The Fourier Transform and Its Applications.* McGraw-Hill, New York, NY, 3rd edition, 1999.
5. Brigham, E. O. *The Fast Fourier Transform and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1988.
6. Chinnery, DG and K. Keutzer. "Closing the gap between ASIC and custom: an ASIC perspective". *Design Automation Conference, 2000. Proceedings 2000. 37th*, 637–642, 2000.
7. Cooley, J.W. and J.W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". *Mathematics of Computation*, 19(90):297–301, 1965.
8. Fabey, Michael. "Revolutionizing Electronic Warfare". Defense News, 8 2005.
9. Gentleman, W.W. and G. Sande. "Fast Fourier transform for fun and profit. Prec. AFIPS 1966 Fall Joint Comput. Conf., Vol. 29".
10. He, S. and M. Torkelson. "A New Approach to Pipeline FFT Processor". *Proc. IEEE Int. Parallel Processing Symp*, 766–770, 1996.
11. He, S. and M. Torkelson. "Design and implementation of a 1024-point pipeline FFT processor". *Custom Integrated Circuits Conference, 1998., Proceedings of the IEEE 1998*, 131–134, 1998.
12. Heo, KL, JH Baek, MH Sunwoo, BG Jo, and BS Son. "New in-place strategy for a mixed-radix FFT processor". *SOC Conference, 2003. Proceedings. IEEE International [Systems-on-Chip]*, 81–84, 2003.
13. Lee, S.Y. and C.C. Chen. "VLSI implementation of programmable FFT architectures for OFDM communication system". *International Conference On Communications And Mobile Computing*, 893–898, 2006.
14. Melnyk, A., B. Dunets, I. Ltd, and U. Lviv. "FFT Processor IP Cores synthesis on the base of configurable pipeline architecture". *CAD Systems in Microelectronics, 2003. CADSM 2003. Proceedings of the 7th International Conference. The Experience of Designing and Application of*, 211–213, 2003.
15. Potipantong, P., T. Wiangtong, P. Sirisuk, and A. Worapishet. "A scaleable FFT/IFFT kernel for communication systems using codesign approach". *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, 329–330, 2005.

16. Press, W. H., B. P. Flannert, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in FORTRAN: The Art of Scientific Computing.* Cambridge University Press, Cambridge, England, 2nd edition, 1989.

17. Saidi, A. "Decimation-in-time-frequency FFT algorithm". *Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, 3, 1994.

18. Soliman, Samir S. and Mandyam D. Srinath. *Continuous and Discrete Signals and Systems.* Prentice Hall, Englewood Cliffs, NJ, first edition, 1990.

19. Stoer, J. and R Bulirsch. *Introduction to Numerical Analysis.* Springer-Verlag, New York, NY, 1980.

20. Sukhsawas, S. and K. Benkrid. "A high-level implementation of a high performance pipeline FFT on Virtex-E FPGAs". *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, 229–232, 2004.

21. Tsui, James. *Digital Techniques for Wideband Receivers.* Artech House, Norwood, MA, second edition, 2001.

22. Uzun, IS, A. Amira, and A. Bouridane. "FPGA implementations of fast Fourier transforms for real-time signal and image processing". *Vision, Image and Signal Processing, IEE Proceedings-*, 152(3):283–296, 2005.

23. Zhao, Y., AT Erdogan, and T. Arslan. "A novel low-power reconfigurable FFT processor". *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, 41–44, 2005.

Vita

Captain Brian Brakus graduated from Hoover High School in North Canton, OH in 1993. He attended the University of Akron as a Computer Engineering student, and graduated in 1999 with a BS in Computer Engineering. After working several civilian jobs for a few years, Captain Brakus joined the US Air Force and was commissioned as an officer in September 2002 through the Officer's Training School.
Capt. Brakus' first assignment was at the National Air and Space Intelligence Center (NASIC) at Wright-Patterson Air Force Base, OH. His following assignment was at the Air Force Institute of Technology (AFIT) to complete a Master's degree in Computer Engineering. He will graduate in March of 2007.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 06–03–2007	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2005 — Mar 2007

4. TITLE AND SUBTITLE

A Modular Mixed Signal
VLSI Design Approach
for
Digital Radar Applications

5a. CONTRACT NUMBER
DACA99–99–C–9999

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Brian M. Brakus, Capt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/07-02

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Dr. Greg Creech, (937)255-4831, email: Greg.Creech@afit.edu
AFRL/SND
2241 Avionics Circle
WPAFB, OH 45433

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This study explores the idea of building a library of VHDL configurable components for use in digital radar applications. Configurable components allows a designer to choose which components he or she needs and configures those components for a specific application. By doing this, design time for ASICs and FPGAs is shortened because the components are already designed and tested. This idea is demonstrated with a configurable dynamic pipelinable fast fourier transform. Many FFT implementations exist, but this implementation is both configurable and dynamic. Pre-synthesis customization allows the FFT to be tailored to almost any DSP application, and the dynamic property allows the FFT to calculate different length FFTs run-time. Three objectives will be accomplished: design and characterization of the aforementioned FFT; analysis of the error involved in the FFT calculation using different twiddle factor bit widths; and finally an analysis of all the configurations for the synthesized design using a 90nm technology library. Speeds of up to 225 MHz have been simulated for a length-1024 FFT using the 90 nm technology.

15. SUBJECT TERMS

FFT, Twiddle Factor, Modular, Library, VHDL, ASIC

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
U	U	U	UU	79	Dr. Yong C. Kim, PhD(ENG)
					19b. TELEPHONE NUMBER *(include area code)* (937) 255–3636; email: yong.kim@afit.edu